Eulerian Cycle (2A)

Young Won Lim
5/19/18

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Path and Trail

A **path** is a **trail** in which all **vertices** are <u>distinct</u>.
(except possibly the first and last)

A **trail** is a **walk** in which all **edges** are <u>distinct</u>.

	Vertices	Edges	
Walk	may repeat	may repeat	(Closed/Open)
Trail	may repeat	<u>cannot</u> repeat	(Open)
Path	<u>cannot</u> repeat	<u>cannot</u> repeat	(Open)
Circuit	may repeat	<u>cannot</u> repeat	(Closed)
Cycle	<u>cannot</u> repeat	<u>cannot</u> repeat	(Closed)

https://en.wikipedia.org/wiki/Eulerian_path

Simple Paths and Cycles

Most literatures require that all of the **edges** and **vertices** of a **path** be distinct from one another.

But, some do not require this and instead use the term **simple path** to refer to a **path** which contains no repeated **vertices**.

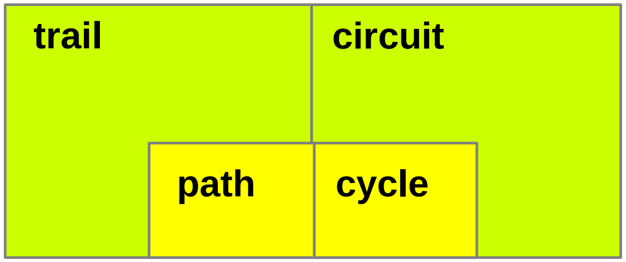
A **simple cycle** may be defined as a **closed walk** with no repetitions of **vertices** and **edges** allowed, other than the repetition of the **starting** and **ending vertex**

There is considerable variation of terminology!!!
Make sure which set of definitions are used...

https://en.wikipedia.org/wiki/Eulerian_path

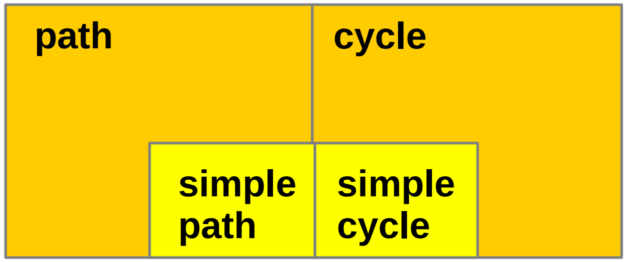
Simple Paths and Cycles

Most literatures



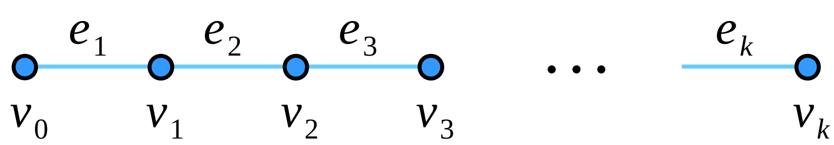
narrow sense path & cycle

some



wide sense path & cycle

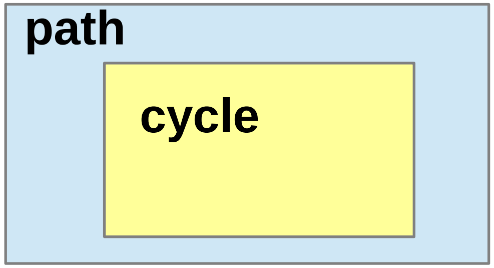
Paths and Cycles



One of a kind

path $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k$

cycle $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k \quad (v_0 = v_k)$



path $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k \quad (v_0 \neq v_k)$

cycle $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k \quad (v_0 = v_k)$

Two different kinds

Euler Cycle

Some people reserve the terms **path** and **cycle** to mean <u>non-self-intersecting</u> path and cycle.	no repeating vertices
A (potentially) <u>self-intersecting</u> path is known as a **trail** or an **open walk**;	repeating vertices
and a (potentially) <u>self-intersecting</u> cycle, a **circuit** or a **closed walk**.	repeating vertices
This ambiguity can be avoided by using the terms **Eulerian trail** and **Eulerian circuit** when <u>self-intersection</u> is allowed	repeating vertices

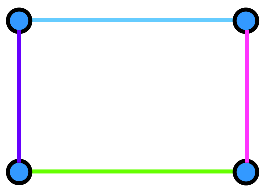
Eulerian ⇒ non-repeating edges

\+ all the edges

https://en.wikipedia.org/wiki/Eulerian_path

Euler Cycle

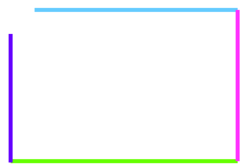
visits every **edge** exactly once



the existence of **Eulerian cycles**

all **vertices** in the graph have an **even** degree

connected graphs with **all vertices** of **even** degree h
ave an **Eulerian cycles**



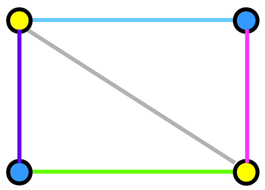
non-repeating edges
repeatable vertices
} **circuit**

Eulerian circuit : more suitable terminology

https://en.wikipedia.org/wiki/Eulerian_path

Euler Path

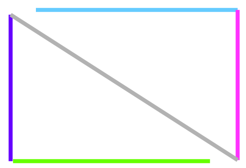
visits <u>every</u> **<u>edge</u>** exactly <u>once</u>



the existence of **Eulerian paths**

all the **vertices** in the graph have an **even** degree

<u>except</u> only **two** vertices with an **odd** degree



An **Eulerian path** starts and ends at <u>different</u> vertices
An **Eulerian cycle** starts and ends at the <u>same</u> vertex.

non-repeating edges
repeatable vertices } **trail**

Eulerian trail : more suitable terminology

https://en.wikipedia.org/wiki/Eulerian_path

Conditions for Eulerian Cycles and Paths

An odd vertex = a vertex with an odd degree
An even vertex = a vertex with an even degree

# of **odd** vertices	Eulerian **Path**	Eulerian **Cycle**
0	No	**Yes**
2	**Yes**	No
4,6,8, …	No	No
1,3,5,7, …	No such graph	No such graph

If the graph is connected

http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

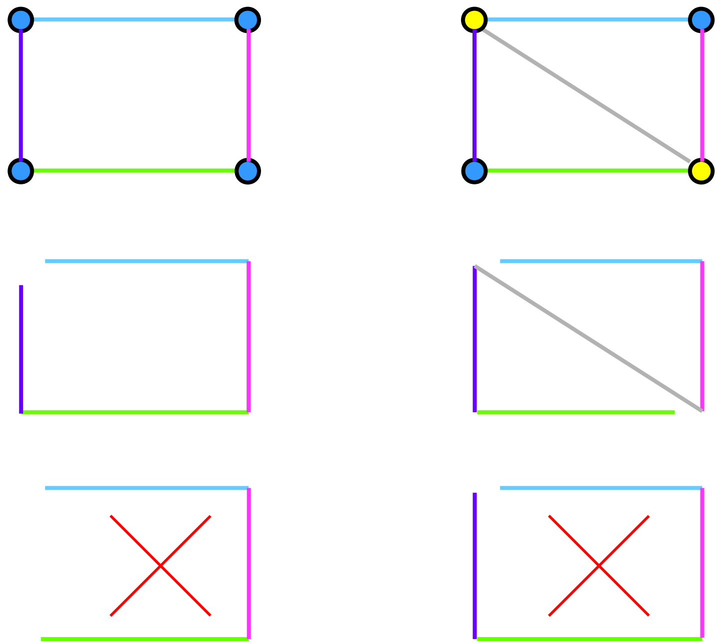
The number of odd vertices

# of **odd** vertices	Eulerian **Path**	Eulerian **Cycle**
0	**No**	**Yes**
2	**Yes**	**No**

of **odd** vertices
= 0

Eulerian Cycle

No Eulerian Path



of **odd** vertices
= 2

Eulerian Path

No Eulerian Cycle

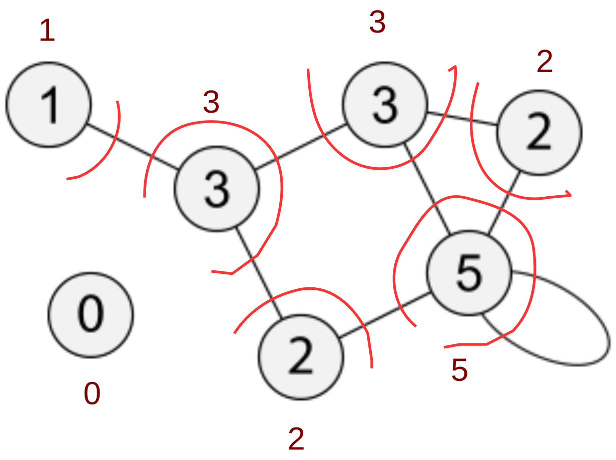
Degree of a vertex

the **degree** (or **valency**) of a vertex is
the number of edges incident to the vertex,
with loops counted twice.

The degree of a vertex v is denoted deg(v)
the maximum degree of a graph G, denoted by Δ(G)
the minimum degree of a graph, denoted by δ(G)

Δ(G) = 5
δ(G) = 0

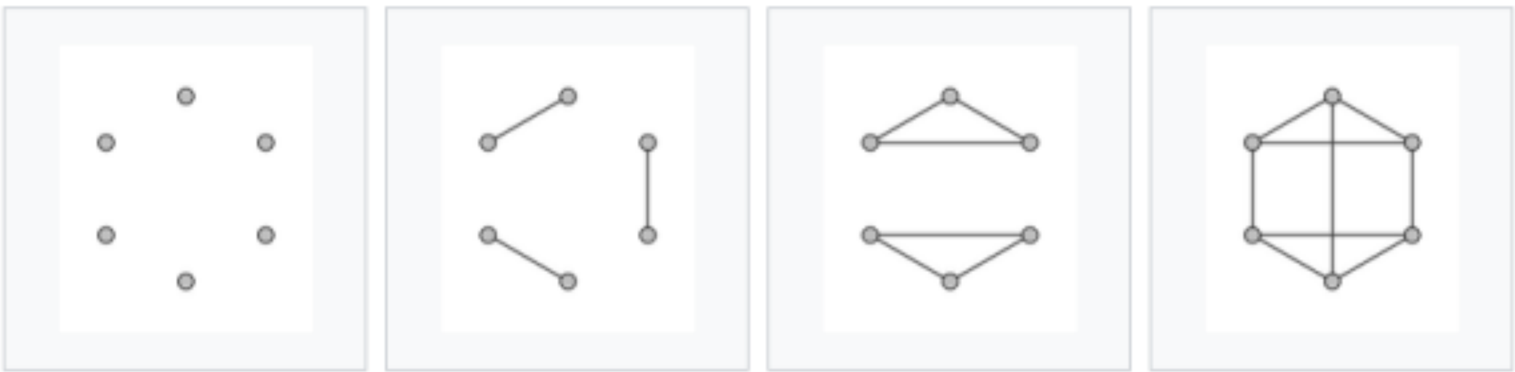
In a **regular** graph, all degrees are the same



https://en.wikipedia.org/wiki/Degree_(graph_theory)

Regular Graphs

a **regular graph** is a graph where each vertex has the same number of neighbors; i.e. every vertex has the same degree or valency.



0-regular graph 1-regular graph 2-regular graph 3-regular graph

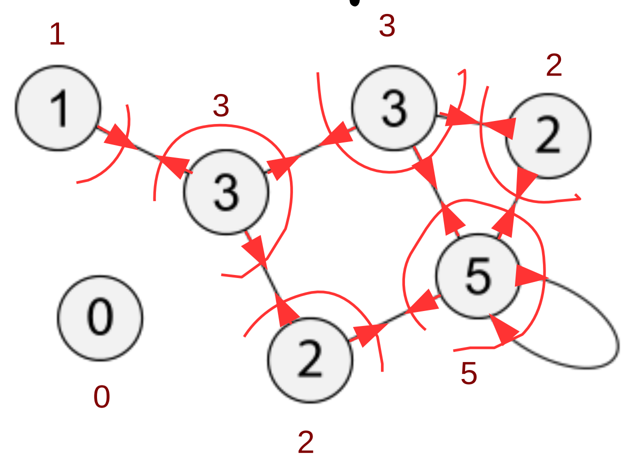
https://en.wikipedia.org/wiki/Regular_graph

Handshake Lemma

E = { edges } |E| = the number of edges

The degree sum formula states that,
given a graph G = (V , E)

$$\sum_{v \in V} \deg(v) = 2|E| \,.$$



The formula implies that in any graph,
the number of vertices with odd degree is even.

This statement (as well as the degree sum formula) is known as the **handshaking lemma**.

# of **odd** vertices	Eulerian **Path**	Eulerian **Cycle**
0	No	Yes
2	Yes	No
4,6,8, ...	No	No
1,3,5,7, ...	No such graph	No such graph

https://en.wikipedia.org/wiki/Degree_(graph_theory)

The number of odd vertices

Odd vertices : $\{x_{1,} x_{2,} \cdots , x_n\}$

$$S = deg(x_1) + deg(x_2) + \cdots + deg(x_n)$$

$$deg(x_i) : even$$

$$S = even + even + \cdots + even$$

Even vertices : $\{y_{1,} y_{2,} \cdots , y_n\}$

$$T = deg(y_1) + deg(y_2) + \cdots + deg(y_n)$$

$$deg(y_i) : odd$$

$$T = odd + odd + \cdots + odd$$

$S : even$

$S+T : even$

$$\Rightarrow T : even = \sum n \; odd \; numbers$$

$$\Rightarrow n : even$$

The formula implies that in any graph,
the number of vertices with odd degree is even.

References

[1] http://en.wikipedia.org/
[2]

Hamiltonian Cycle (3A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Hamiltonian Cycles – Properties (3)

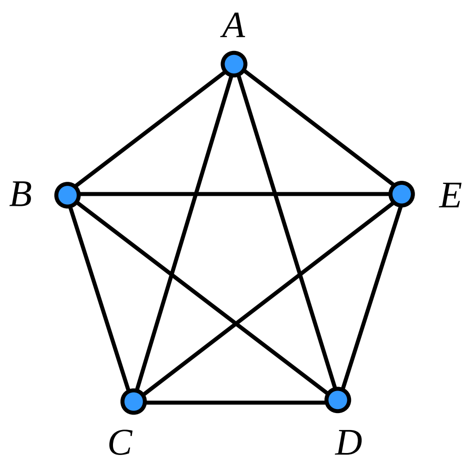
A tournament (with more than two vertices) is Hamiltonian if and only if it is **strongly connected**.

The number of different Hamiltonian cycles
in a **complete undirected** graph on **n** vertices is **(n − 1)! / 2**
in a complete directed graph on n vertices is (n − 1)!.

These counts assume that cycles that are the same apart from their starting point are not counted separately.

https://en.wikipedia.org/wiki/Hamiltonian_path

Number of Hamiltonian Cycles (1)

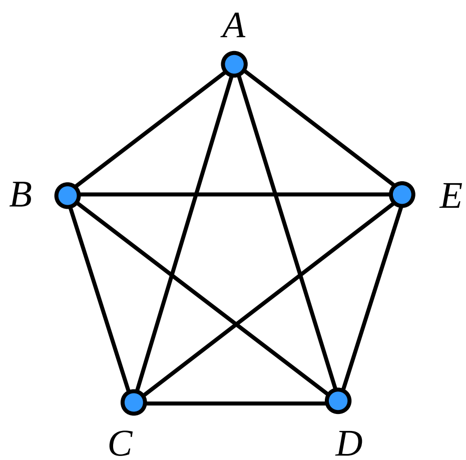


$$(5-1)! = 24$$

https://en.wikipedia.org/wiki/Hamiltonian_path

ABCDE	*BACDE*	*CABDE*	*DABCE*	*EABCD*
ABCED	*BACED*	*CABED*	*DABEC*	*EABDC*
ABDCE	*BADCE*	*CADBE*	*DACBE*	*EACBD*
ABDEC	*BADEC*	*CADEB*	*DACEB*	*EACDB*
ABECD	*BAECD*	*CAEBD*	*DADBC*	*EADBC*
ABEDC	*BAEDC*	*CAEDB*	*DADCB*	*EADCB*
ACBDE	*BCADE*	*CBADE*	*DBACE*	*EBACD*
ACBED	*BCAED*	*CBAED*	*DBAEC*	*EBADC*
ACDBE	*BCDAE*	*CBDAE*	*DBCAE*	*EBCAD*
ACDEB	*BCDEA*	*CBDEA*	*DBCEA*	*EBCDA*
ACEBD	*BCEAD*	*CBEAD*	*DBEAC*	*EBDAC*
ACEDB	*BCEDA*	*CBEDA*	*DBECA*	*EBDCA*
ADBCE	*BDACE*	*CDABE*	*DCABE*	*ECABD*
ADBEC	*BDAEC*	*CDAEB*	*DCAEB*	*ECADB*
ADCBE	*BDCAE*	*CDBAE*	*DCBAE*	*ECBAD*
ADCEB	*BDCEA*	*CDBEA*	*DCBEA*	*ECBDA*
ADEBC	*BDEAC*	*CDEAB*	*DCEAB*	*ECDAB*
ADECB	*BDECA*	*CDEBA*	*DCEBA*	*ECDBA*
AEBCD	*BEACD*	*CEABD*	*DEABC*	*EDABC*
AEBDC	*BEADC*	*CEADB*	*DEACB*	*EDACB*
AECBD	*BECAD*	*CEBAD*	*DEBAC*	*EDBAC*
AECDB	*BECDA*	*CEBDA*	*DEBCA*	*EDBCA*
AEDBC	*BEDAC*	*CEDAB*	*DECAB*	*EDCAB*
AEDCB	*BEDCA*	*CEDBA*	*DECBA*	*EDCBA*

Number of Hamiltonian Cycles (2)



$$(5-1)! = 24$$

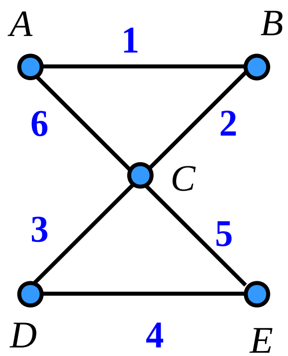
https://en.wikipedia.org/wiki/Hamiltonian_path

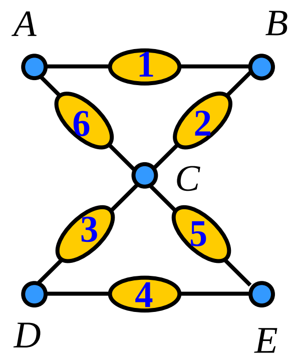
A	BCDE	AB	CDE	ABC	DE	ABCD	E	ABCDE
				ABD	CE	ABCE	D	ABCED
		AC	BDE	ABE	CD	ABDC	E	ABDCE
						ABDE	C	ABDEC
		AD	BCE	ACB	DE	ABEC	D	ABECD
				ACD	BE	ABED	C	ABEDC
		AE	BCD	ACE	BD			
						ACBD	E	ACBDE
				ADB	CE	ACBE	D	ACBED
				ADC	BE	ACDB	E	ACDBE
				ADE	BC	ACDE	B	ACDEB
						ACEB	D	ACEBD
				AEB	CD	ACED	B	ACEDB
				AEC	BD			
				AED	BC	ADBC	E	ADBCE
						ADBE	C	ADBEC
						ADCB	E	ADCBE
						ADCE	B	ADCEB
						ADEB	C	ADEBC
						ADEC	B	ADECB
						AEBC	D	AEBCD
						AEBD	C	AEBDC
						AECB	D	AECBD
						AECD	B	AECDB
						AEDB	C	AEDBC
						AEDC	B	AEDCB

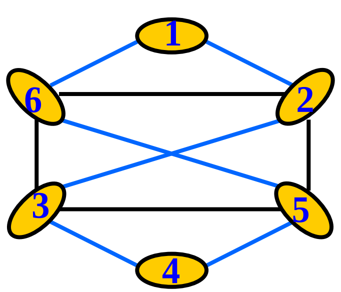
Eulerian Graph (1)

The **Eulerian cycle** corresponds to a **Hamiltonian cycle** in the **line graph L(G)**, so the **line graph** of every **Eulerian graph** is **Hamiltonian graph**.

G → **L(G)**







Eulerian Cycle
ABCDECA

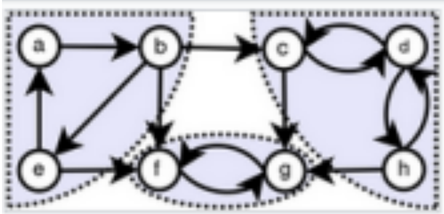
→

Hamiltonian Cycle
1-2-3-4-5-6-1

Strongly Connected Component

a directed graph is said to be **strongly connected** or **diconnected** if every **vertex** is reachable from every other **vertex.**

The **strongly connected components** or **diconnected components** of an arbitrary directed graph form a **partition** into **subgraphs** that are themselves **strongly connected**.

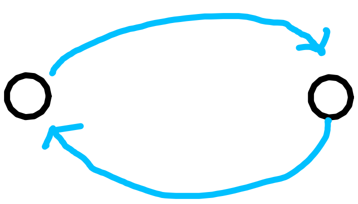


Graph with strongly connected components marked

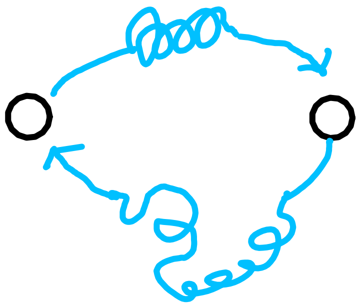
https://en.wikipedia.org/wiki/Hamiltonian_path

SCC and WCC

a directed graph is **strongly connected**
if there is a **path** from **a** to **b** and from **b** to **a**
whenever **a** and **b** are **vertices** in the graph

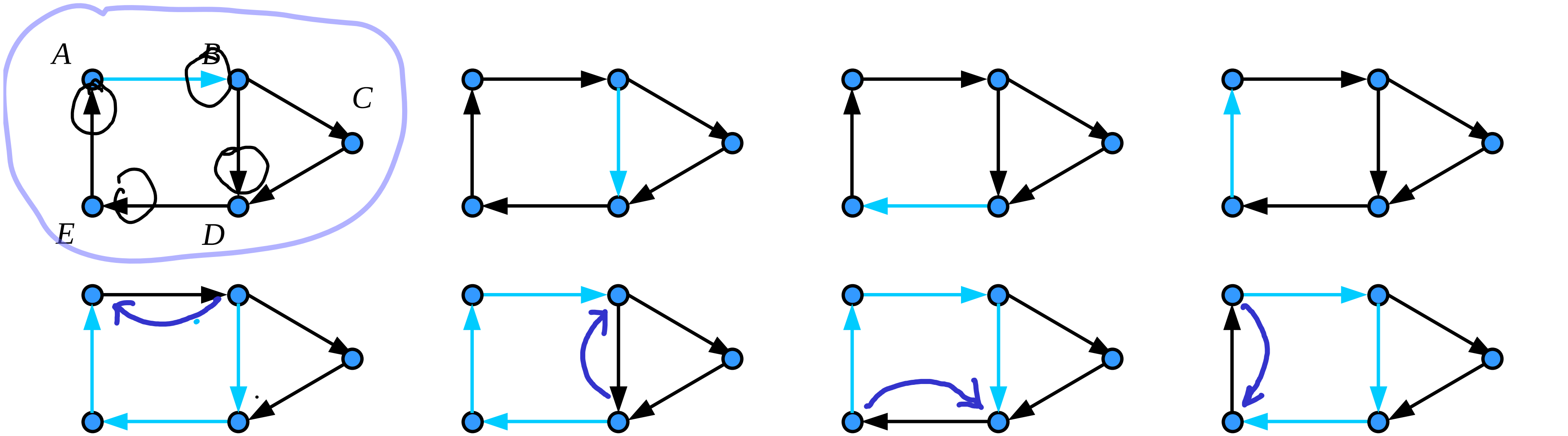


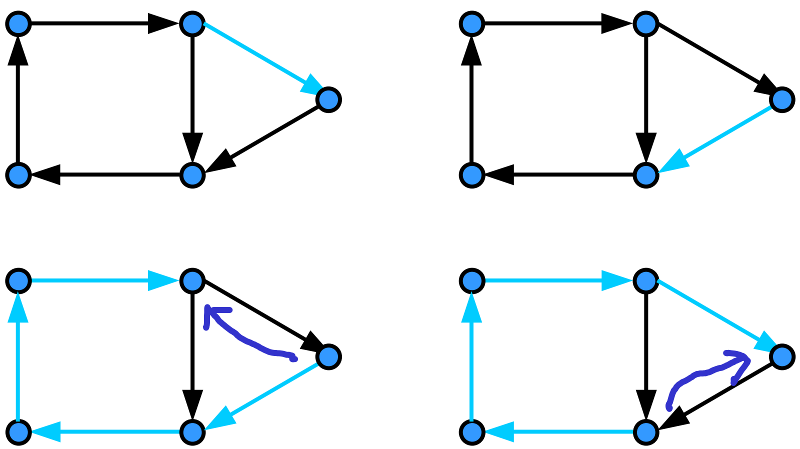
a directed graph is **weakly connected**
if there is a **path** between every two **vertices**
in the underlying undirected graph
(either way)
directions of edges are disregarded



Discrete Mathematics, Rosen

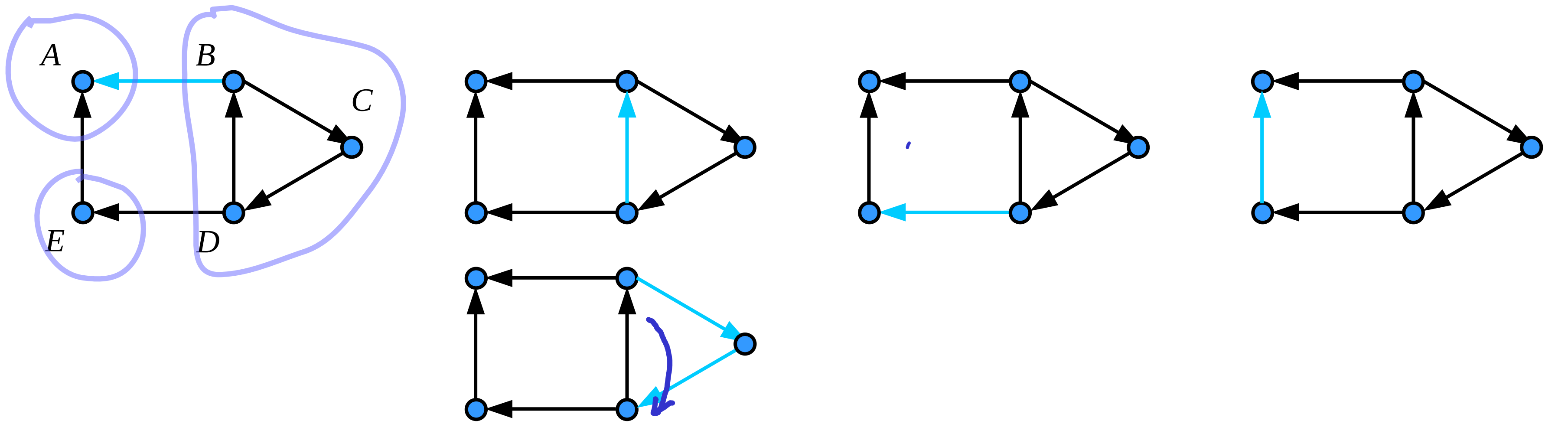
SC examples (1)

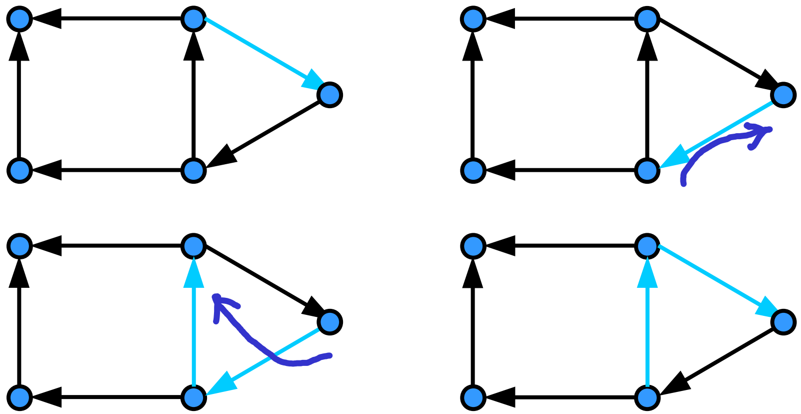




Discrete Mathematics, Rosen

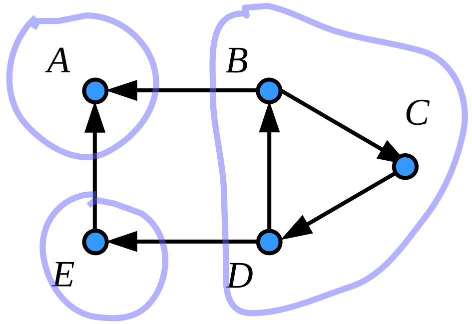
SC examples (2)



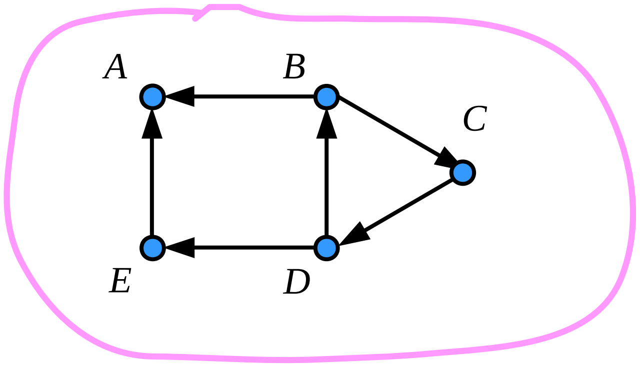


Discrete Mathematics, Rosen

SCC and WCC examples



three strongly connected components



one weakly connected components

Discrete Mathematics, Rosen

References

[1] http://en.wikipedia.org/
[2]

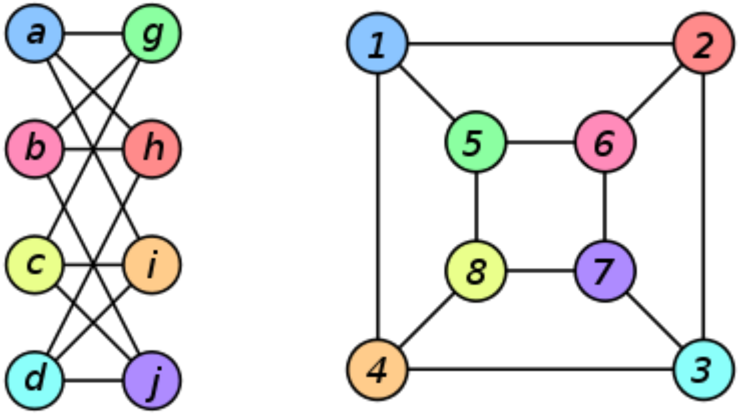
Isomorphic Graph (8A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice and Octave.

Graph Isomorphism

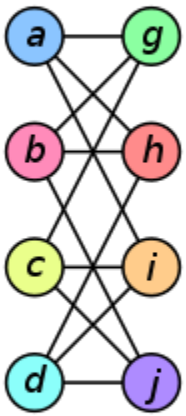
The two graphs shown below are **isomorphic**, despite their different looking drawings.



f(a) = 1
f(b) = 6
f(c) = 8
f(d) = 3
f(g) = 5
f(h) = 2
f(i) = 4
f(j) = 7

https://en.wikipedia.org/wiki/Graph_isomorphism

Graph G_1 and its Adjacency Matrix

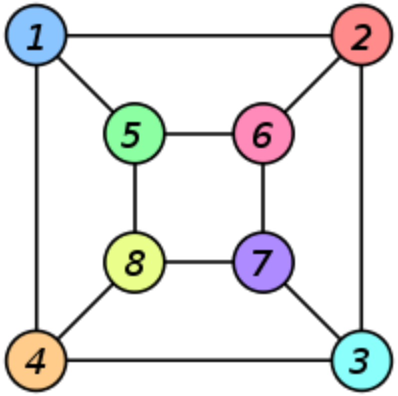


	a	b	c	d	g	h	i	j
a	0	0	0	0	1	1	1	0
b	0	0	0	0	1	1	0	1
c	0	0	0	0	1	0	1	1
d	0	0	0	0	0	1	1	1
g	1	1	1	0	0	0	0	0
h	1	1	0	1	0	0	0	0
i	1	0	1	1	0	0	0	0
j	0	1	1	1	0	0	0	0

https://en.wikipedia.org/wiki/Graph_isomorphism

Graph G_2 and its Adjacency Matrix

	1	2	3	4	5	6	7	8
1	0	1	0	1	1	0	0	0
2	1	0	1	0	0	1	0	0
3	0	1	0	1	0	0	1	0
4	1	0	1	0	0	0	0	1
5	1	0	0	0	0	1	0	1
6	0	1	0	0	1	0	1	0
7	0	0	1	0	0	1	0	1
8	0	0	0	1	1	0	1	0

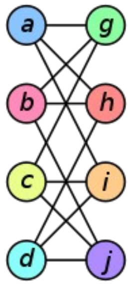


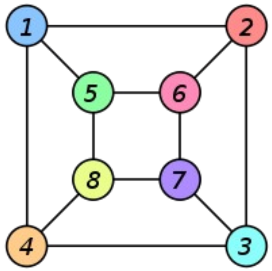
edge-preserving bijection

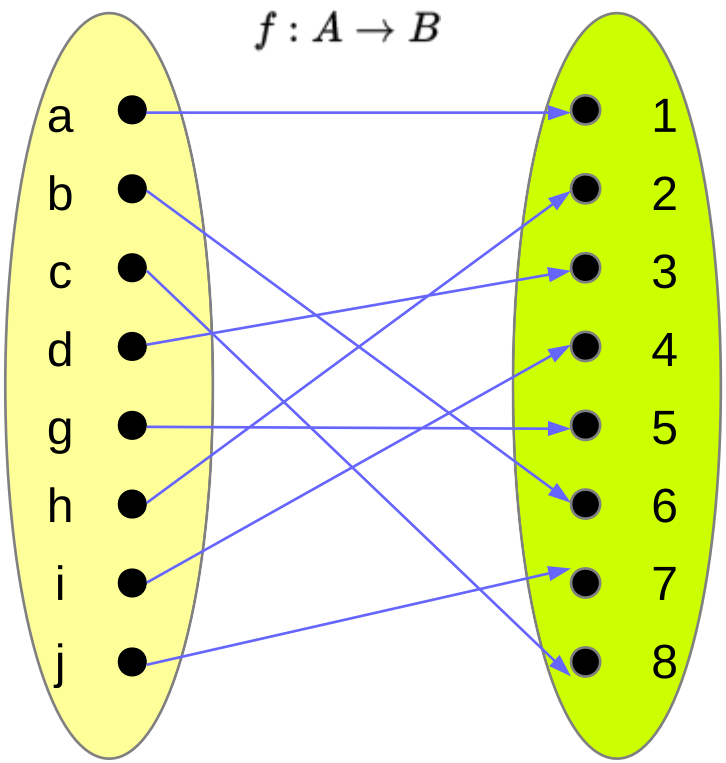
structure-preserving bijection.

https://en.wikipedia.org/wiki/Graph_isomorphism

Bijection Mapping f







		1	6	8	3	5	2	4	7
		a	b	c	d	g	h	i	j
1	a	0	0	0	0	1	1	1	0
6	b	0	0	0	0	1	1	0	1
8	c	0	0	0	0	1	0	1	1
3	d	0	0	0	0	0	1	1	1
5	g	1	1	1	0	0	0	0	0
2	h	1	1	0	1	0	0	0	0
4	i	1	0	1	1	0	0	0	0
7	j	0	1	1	1	0	0	0	0

Converting the Adjacency Matrix

permuting the rows and columns

	1	6	8	3	5	2	4	7
1	0	0	0	0	1	1	1	0
6	0	0	0	0	1	1	0	1
8	0	0	0	0	1	0	1	1
3	0	0	0	0	0	1	1	1
5	1	1	1	0	0	0	0	0
2	1	1	0	1	0	0	0	0
4	1	0	1	1	0	0	0	0
7	0	1	1	1	0	0	0	0

Adjacency Matrix of G_1

	1	2	3	4	5	6	7	8
1	0	1	0	1	1	0	0	0
2	1	0	1	0	0	1	0	0
3	0	1	0	1	0	0	1	0
4	1	0	1	0	0	0	0	1
5	1	0	0	0	0	1	0	1
6	0	1	0	0	1	0	1	0
7	0	0	1	0	0	1	0	1
8	0	0	0	1	1	0	1	0

Adjacency Matrix of G_2

Converting the Adjacency Matrix

	1	6	8	3	5	2	4	7
1	0	0	0	0	1	1	1	0
6	0	0	0	0	1	1	0	1
8	0	0	0	0	1	0	1	1
3	0	0	0	0	0	1	1	1
5	1	1	1	0	0	0	0	0
2	1	1	0	1	0	0	0	0
4	1	0	1	1	0	0	0	0
7	0	1	1	1	0	0	0	0

	1	2	3	4	5	6	7	8
1	0	1	0	1	1	0	0	0
6	0	1	0	0	1	0	1	0
8	0	0	0	1	1	0	1	0
3	0	1	0	1	0	0	1	0
5	1	0	0	0	0	1	0	1
2	1	0	1	0	0	1	0	0
4	1	0	1	0	0	0	0	1
7	0	0	1	0	0	1	0	1

	1	2	3	4	5	6	7	8
1	0	1	0	1	1	0	0	0
2	1	0	1	0	0	1	0	0
3	0	1	0	1	0	0	1	0
4	1	0	1	0	0	0	0	1
5	1	0	0	0	0	1	0	1
6	0	1	0	0	1	0	1	0
7	0	0	1	0	0	1	0	1
8	0	0	0	1	1	0	1	0

G_1 adjacency matrix after maping

G_2 adjacency matrix after permuting rows and columns

References

[1] http://en.wikipedia.org/
[2]

Planar Graph (7A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Planar Graph

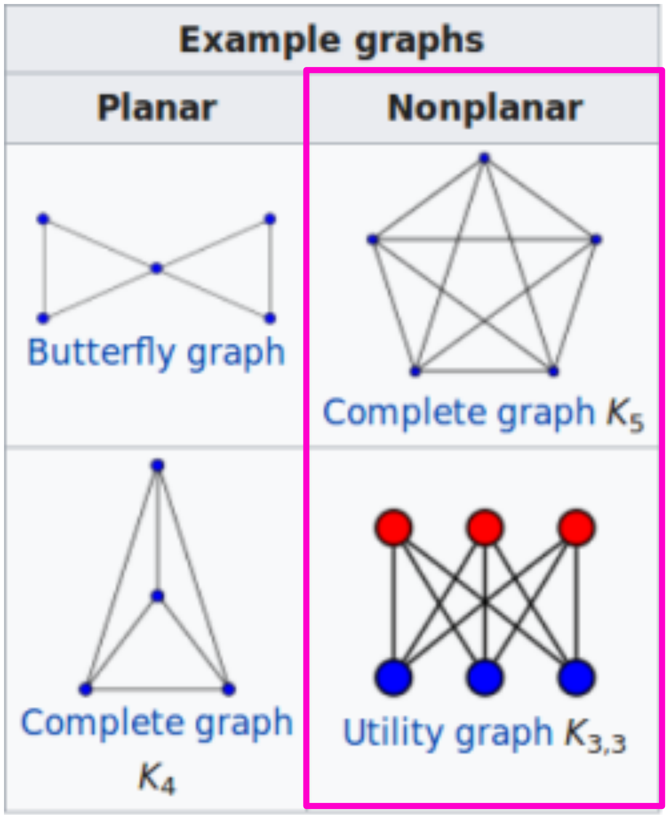
a planar graph is a graph that can be embedded in the plane, i.e., it can be drawn on the plane in such a way that its edges intersect only at their endpoints.

it can be drawn in such a way that no edges cross each other. Such a drawing is called a **plane graph** or **planar embedding** of the graph. (**planar representation**)

A **plane graph** can be defined as a planar graph with a mapping from every node to a point on a plane, and from every edge to a plane curve on that plane,
such that the extreme points of each curve are the points mapped from its end nodes, and all curves are disjoint except on their extreme points.

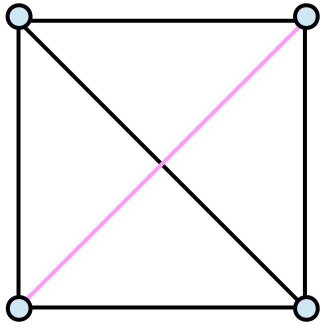
https://en.wikipedia.org/wiki/Planar_graph

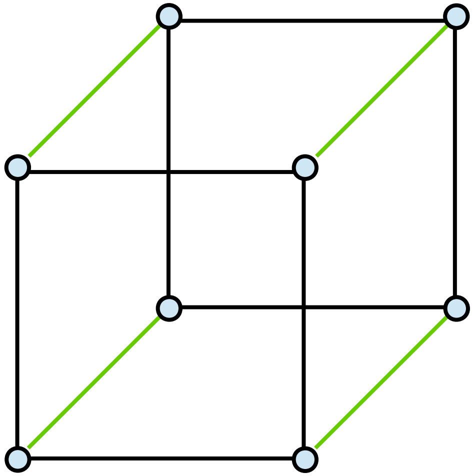
Planar Graph Examples

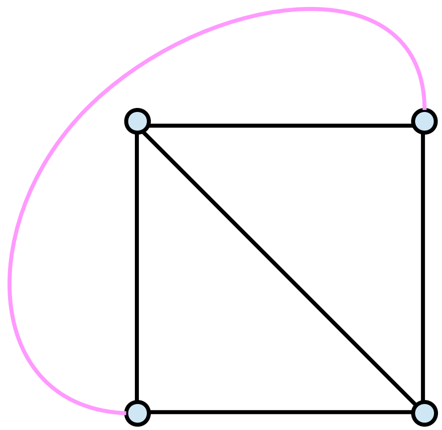


https://en.wikipedia.org/wiki/Planar_graph

Planar Representation

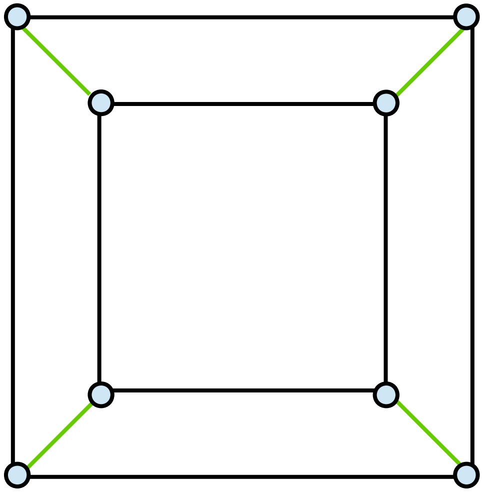






No crossing

K_4 Planar

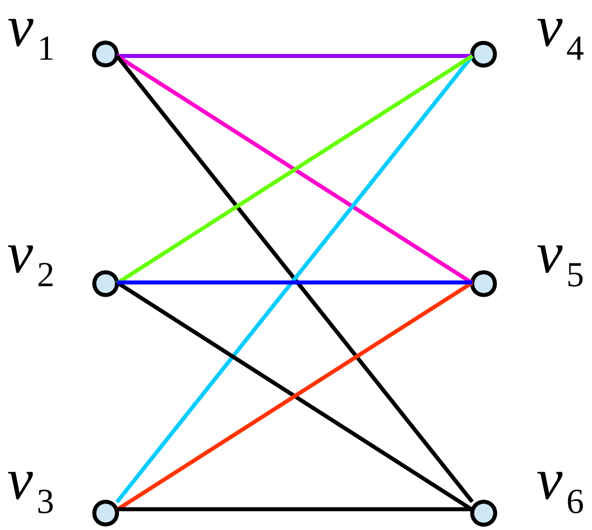


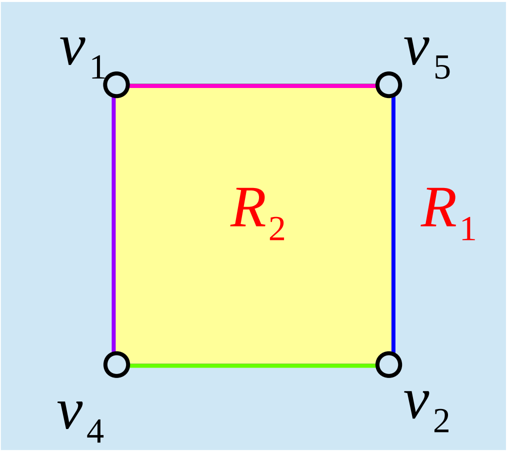
No crossing

Q_3 Planar

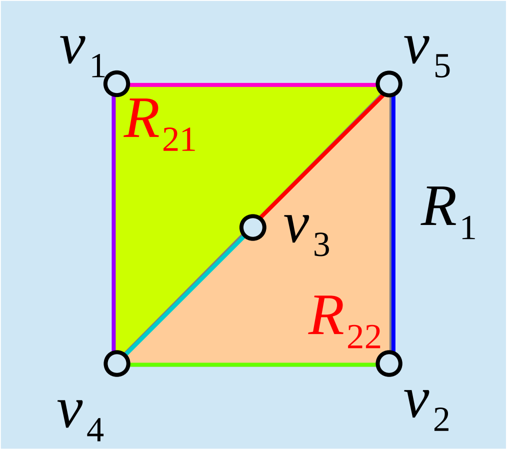
Discrete Mathematics, Rosen

Non-planar Graph $K_{3,3}$





no where v_6



Non-planar

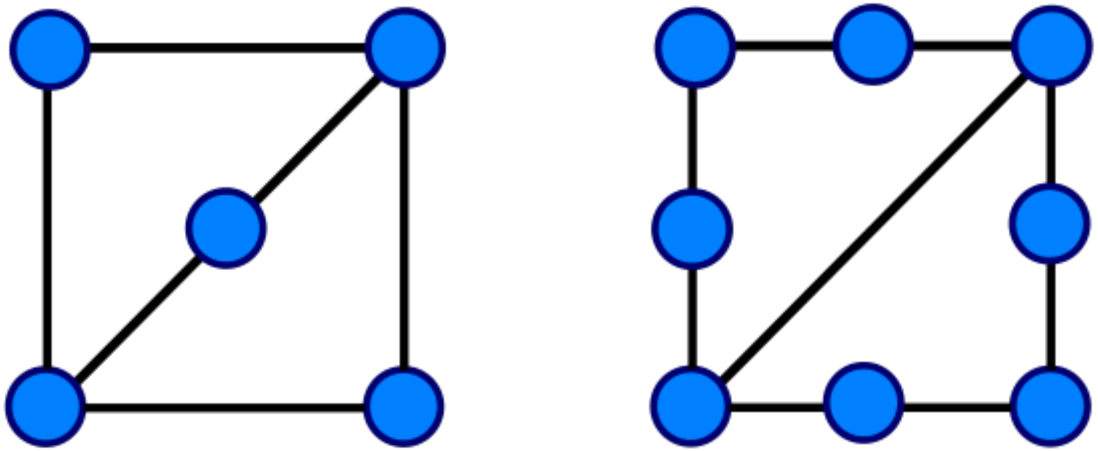
Discrete Mathematics, Rosen

Homeomorphism

two graphs G and G′ are **homeomorphic**
if there is a graph **isomorphism**
from some **subdivision** of G
to some **subdivision** of G′.

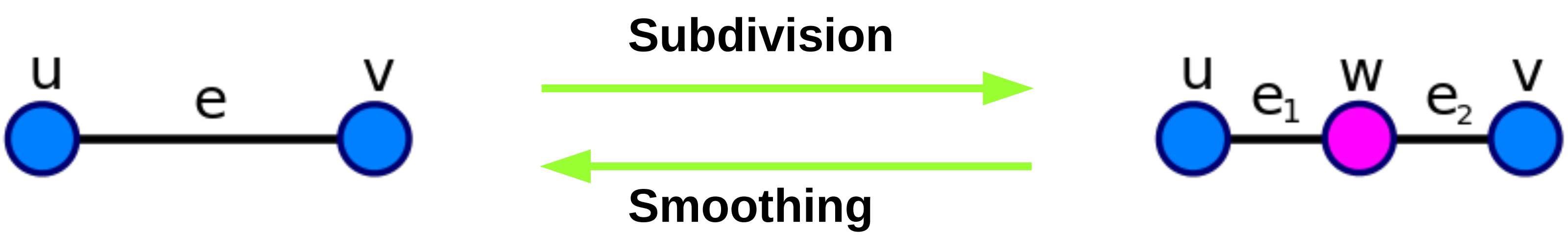
homeo (identity, sameness)

iso (equal)



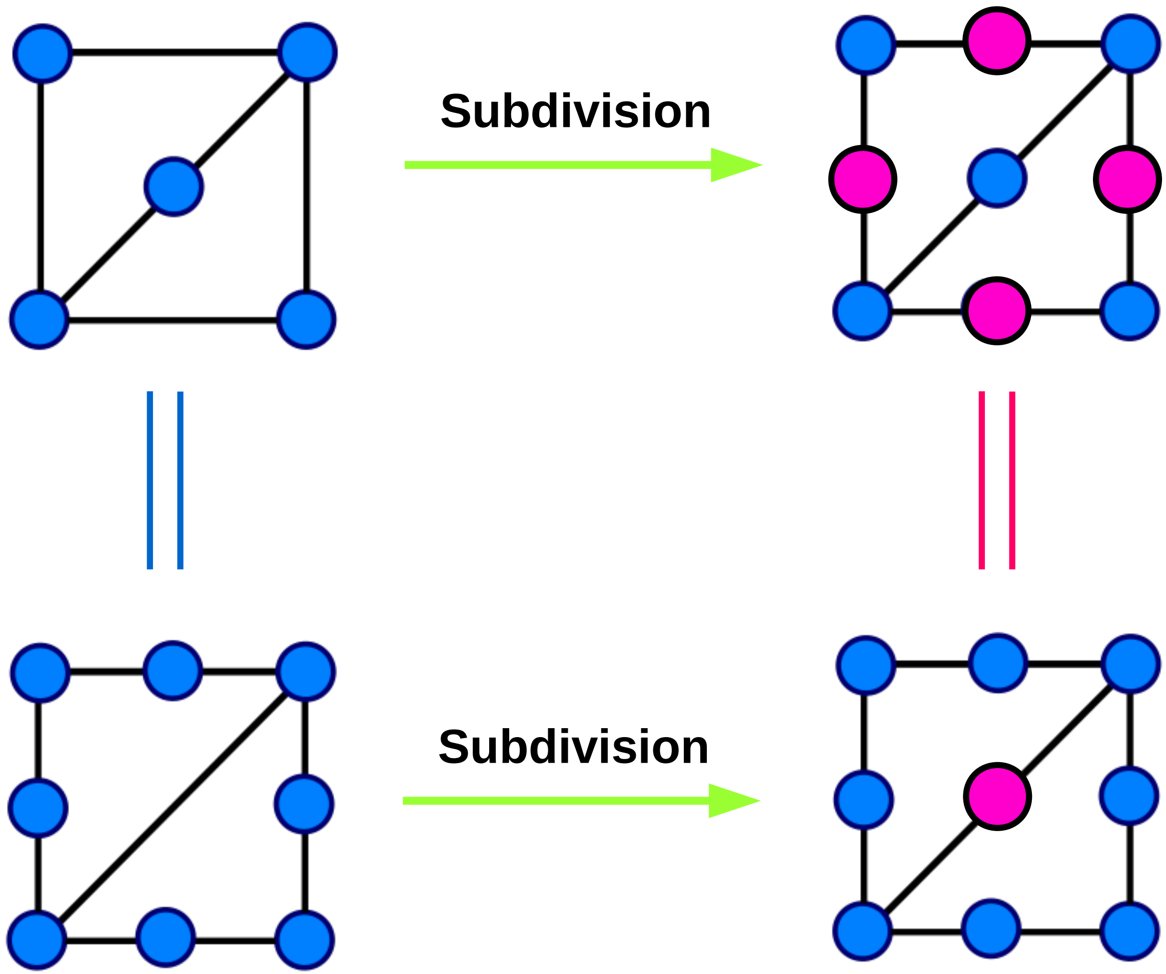
https://en.wikipedia.org/wiki/Planar_graph

Subdivision and Smoothing



https://en.wikipedia.org/wiki/Planar_graph

Homeomorphism Examples



https://en.wikipedia.org/wiki/Planar_graph

homeomorphic —Subdivision→ isomorphic

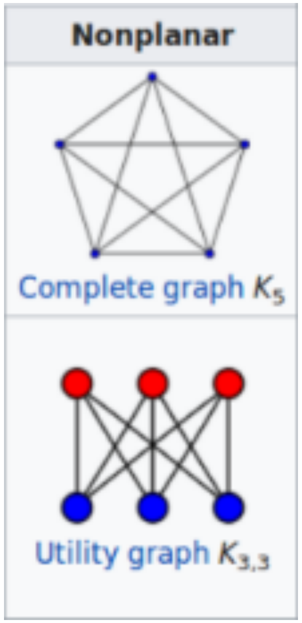
Embedding on a surface

subdividing a graph preserves **planarity**.

Kuratowski's theorem states that

> a finite graph is **planar** if and only if
> it contains **no** subgraph **homeomorphic**
> to $\mathbf{K_5}$ (complete graph on five vertices) or
> $\mathbf{K_{3,3}}$ (complete bipartite graph on six vertices,
> three of which connect to each of the other three).

In fact, a graph **homeomorphic** to $\mathbf{K_5}$ or $\mathbf{K_{3,3}}$
is called a **Kuratowski subgraph**.

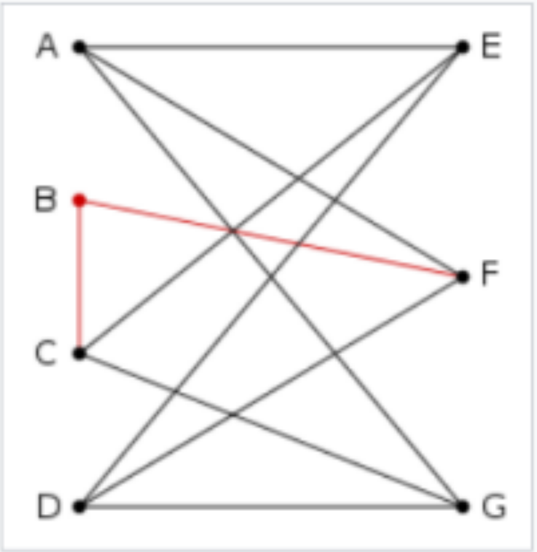


https://en.wikipedia.org/wiki/Planar_graph

Kuratowski's Theorem

A finite graph is planar if and only if
it does not contain a **subgraph**
that is a **subdivision** of the complete graph $\mathbf{K_5}$ or
the complete bipartite graph $\mathbf{K_{3,3}}$ (utility graph).

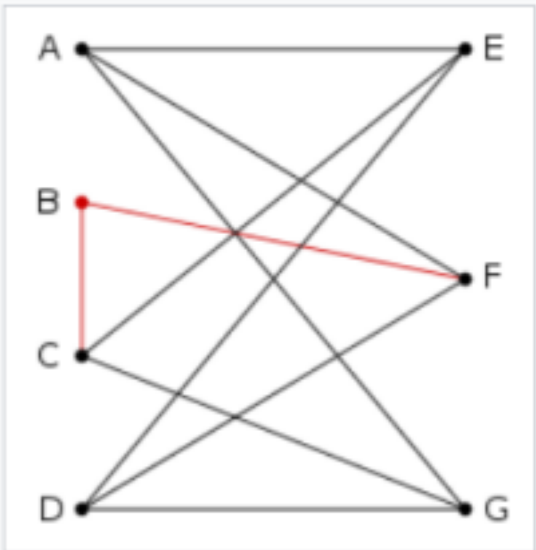
A subdivision of a graph results
from inserting vertices into edges
(changing an edge •——• to •—•—•)
zero or more times.



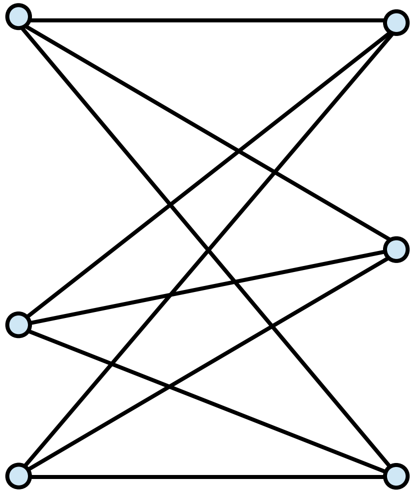
An example of a graph with no K_5 or $K_{3,3}$ subgraph. However, it contains a subdivision of $K_{3,3}$ and is therefore non-planar.

https://en.wikipedia.org/wiki/Planar_graph

Kuratowski’s Theorem

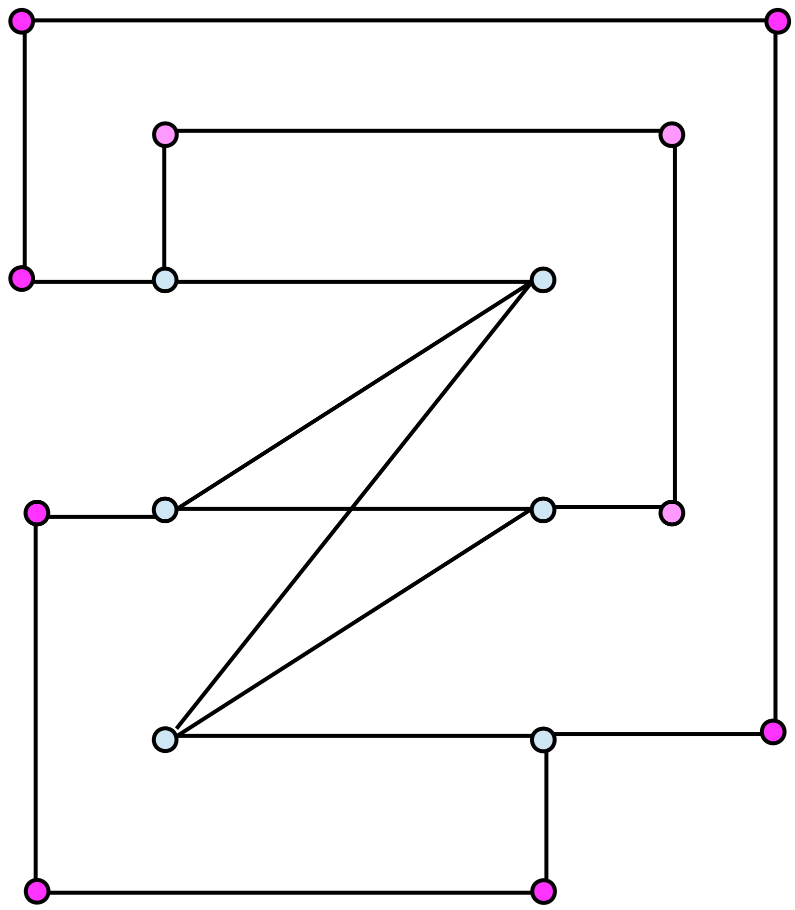


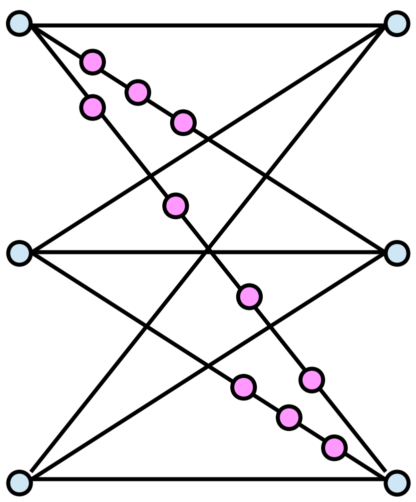
An example of a graph with no K_5 or $K_{3,3}$ subgraph. However, it contains a subdivision of $K_{3,3}$ and is therefore non-planar.



https://en.wikipedia.org/wiki/Planar_graph

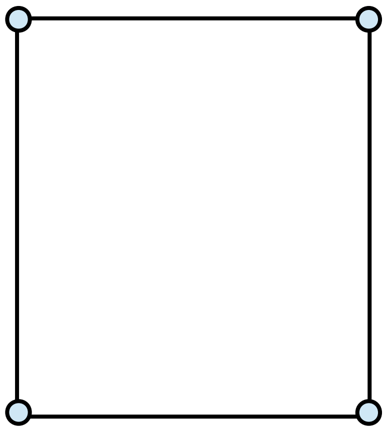
A subdivision of $K_{3,3}$



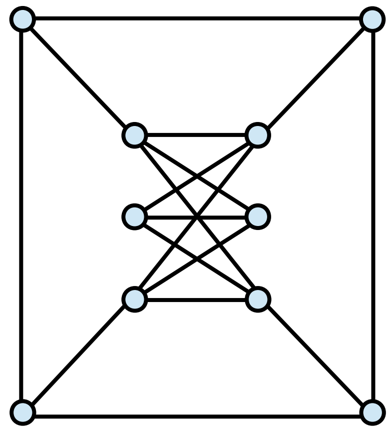


Non-planar graph examples

Planar

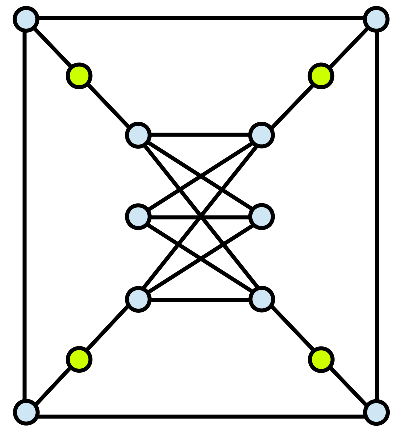


Non-planar



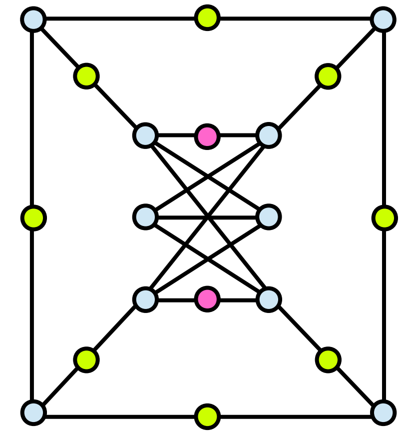
contains $K_{3,3}$

Non-planar



contains $K_{3,3}$

Non-planar



contains a subdivision of $K_{3,3}$

Euler’s Formula

Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then

$$v - e + f = 2$$

https://en.wikipedia.org/wiki/Planar_graph

Euler’s Formula

In a finite, connected, simple, planar graph, any face (except possibly the outer one) is bounded by at least three edges and every edge touches at most two faces; using Euler's formula, one can then show that these graphs are sparse in the sense that if $v \geq 3$:

$$e \leq 3v - 6$$

https://en.wikipedia.org/wiki/Planar_graph

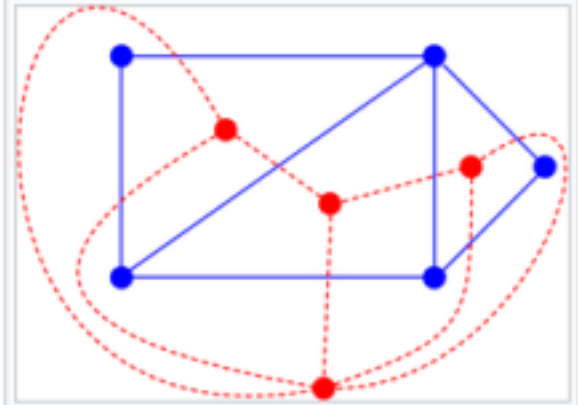
Dual Graph

the dual graph of a plane graph G is a graph that has a **vertex** for each **face** of G.

The dual graph has an **edge** whenever two **faces** of G are separated from each other by an **edge**,

and a **self-loop** when the same **face** appears on both sides of an **edge**.

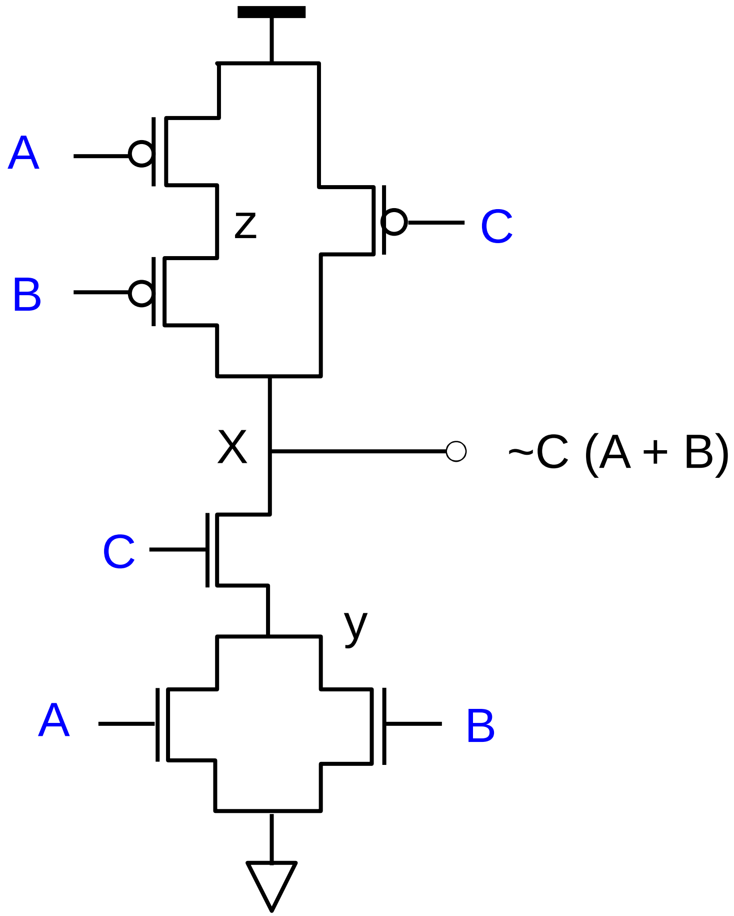
each **edge e** of G has a corresponding **dual edge**, whose endpoints are the **dual vertices** corresponding to the **faces** on either side of **e**.

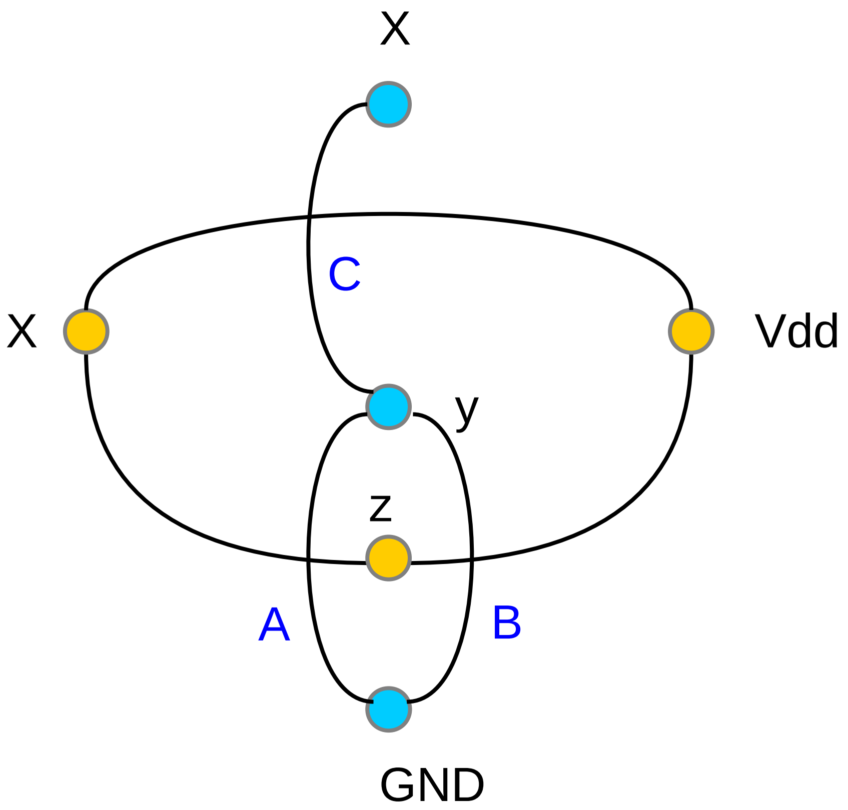


The red graph is the dual graph of the blue graph, and *vice versa*.

https://en.wikipedia.org/wiki/Hamiltonian_path

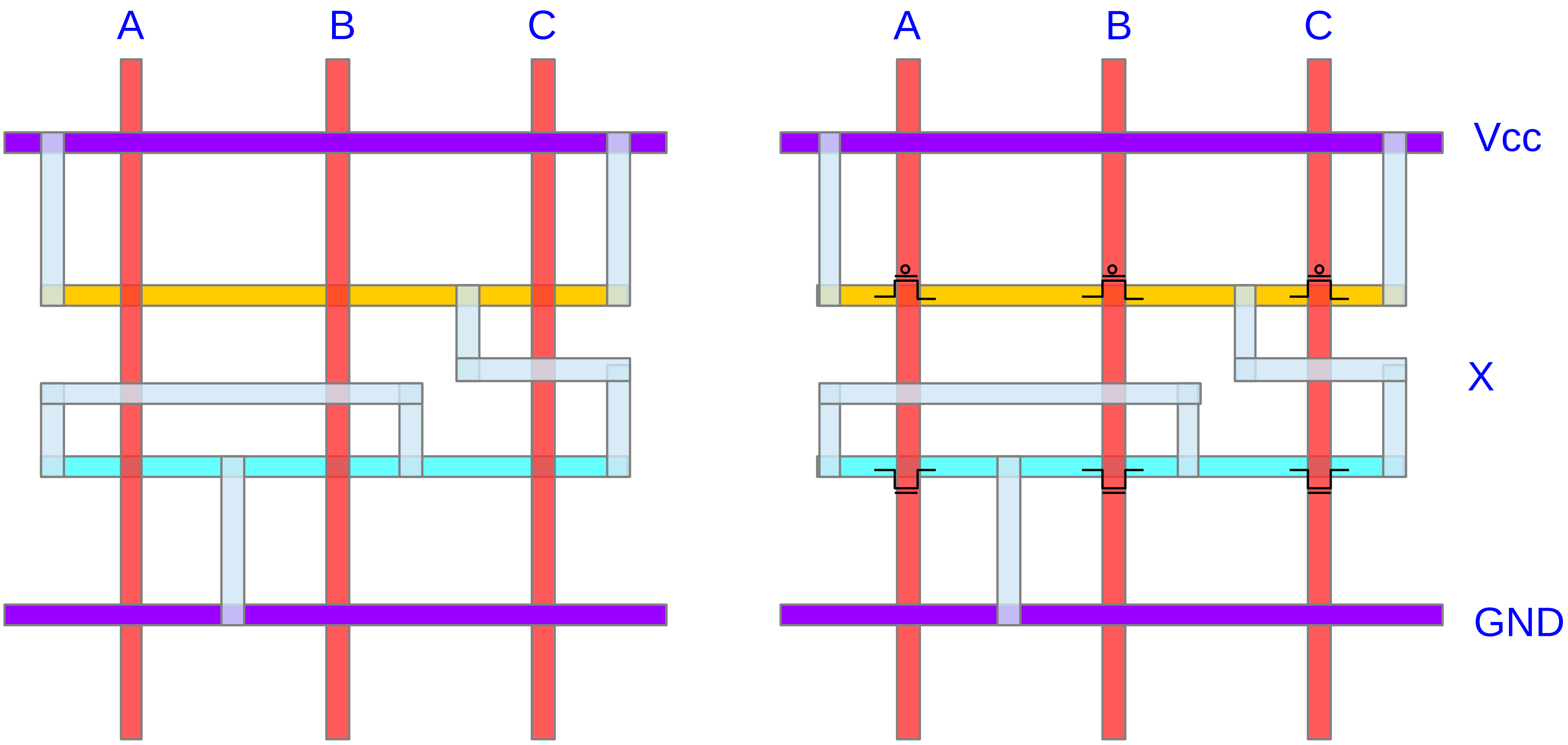
Dual Graph





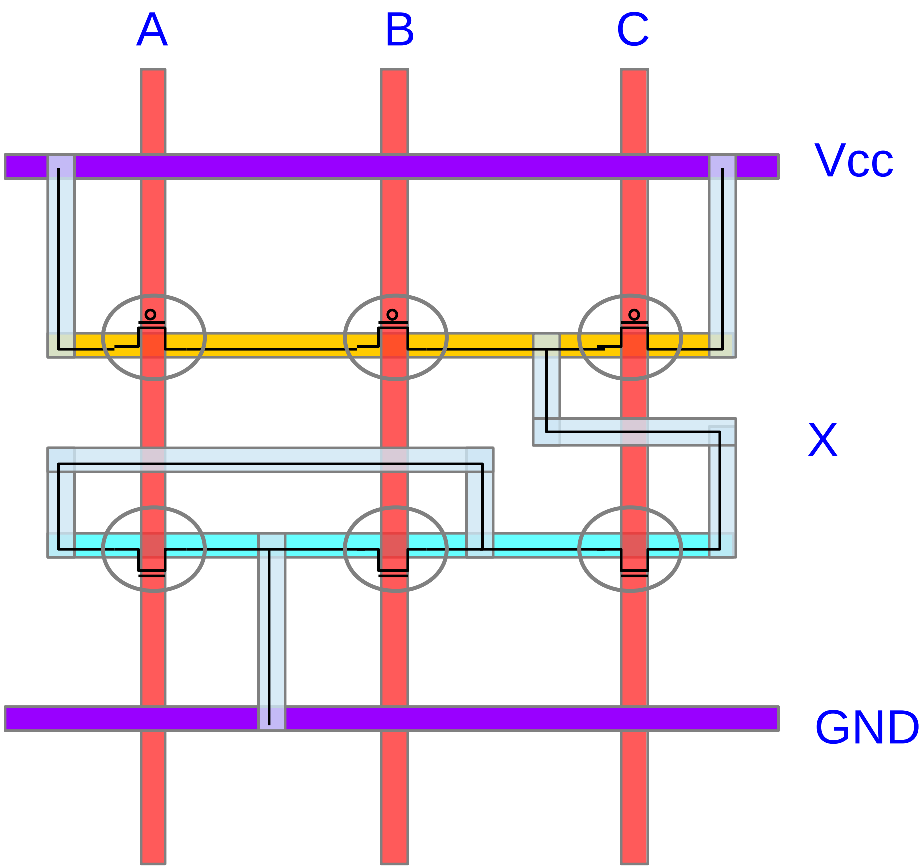
http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

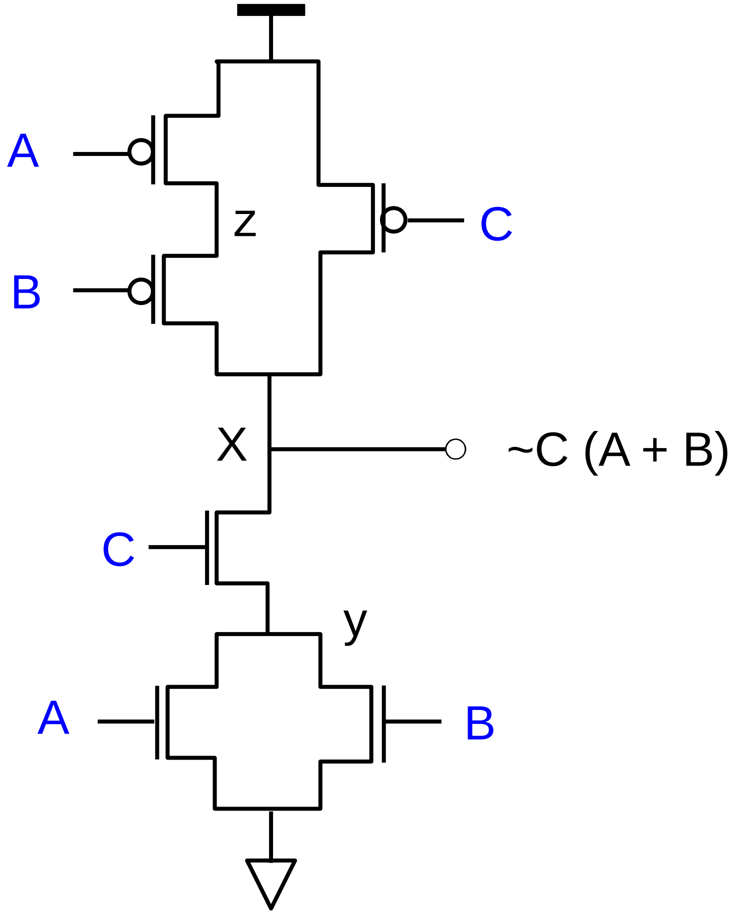
Stick Layout



http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

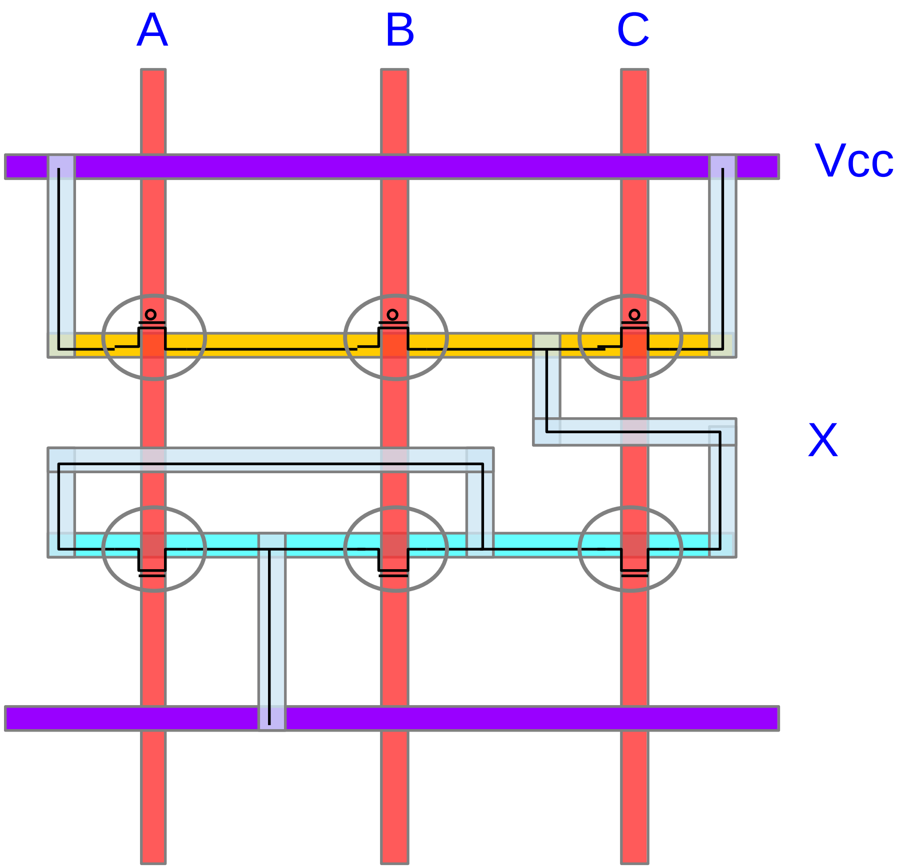
Stick Graph and Logic Diagram



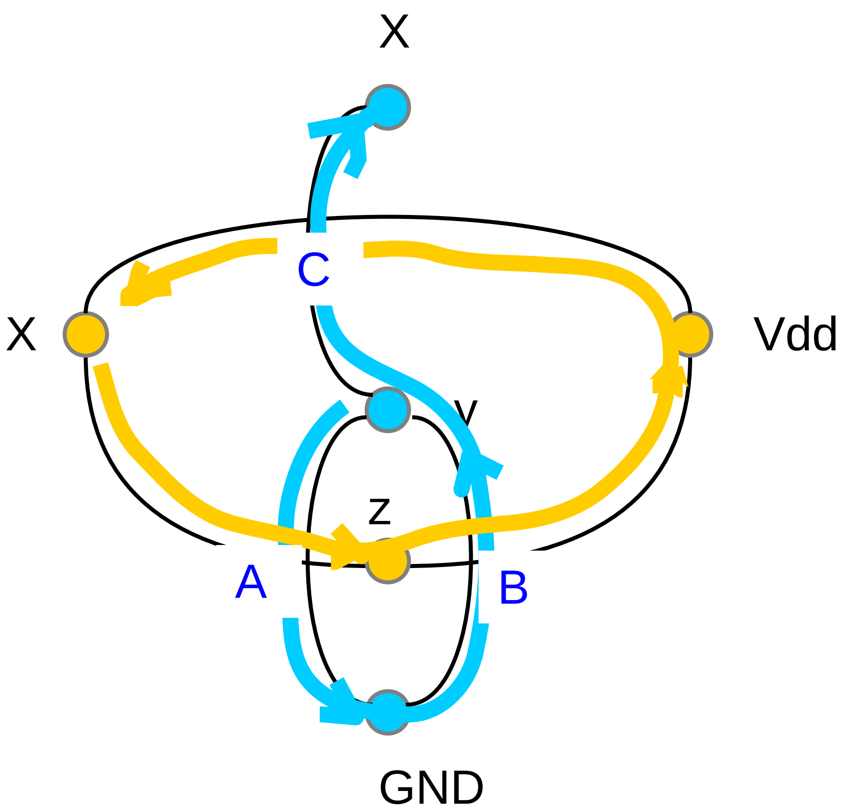


http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

Stick Graph and Logic Diagram



uninterrupted diffusion strip



consistent Euler paths (PUN & PDN)

http://www.cse.psu.edu/~kxc104/class/cmpen411/11s/lec/C411L06StaticLogic.pdf

References

[1] http://en.wikipedia.org/
[2]

Graph Search (6A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Graph Traversal

graph traversal (**graph search**) refers to
the process of visiting (checking and/or updating)
each **vertex** in a graph.

Such traversals are classified
by the order in which the vertices are visited.

Tree traversal is a special case of **graph traversal**.

https://en.wikipedia.org/wiki/Graph_traversal

General Graph Search Algorithm

```
Search( start, isGoal, criteria)
    insert(Start, Open);
    repeat
    if (empty(Open)) then return fail;
    select node from Open using Criteria;
    mark node as visited;
    if (isGoal(node)) then return node;
```

https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

DFS

Open – Stack
Criteria – **pop**

```
DFS( Start, isGoal)
    push(Start, Open);
    repeat
        if (empty(Open)) then return fail;
        node := pop(Open);
        Mark node as visited;
        if (isGoal(node)) then return node;
        for each child of node do
            if (child not already visited) then
                push(child, Open);
```

https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

BFS

Open – Stack
Criteria – **dequeue**

```
BFS( Start, isGoal)
    enqueue(Start, Open);
    repeat
        if (empty(Open)) then return fail;
        node := dequeue(Open);
        mark node as visited;
        if (isGoal(node)) then return node;
        for each child of node do
            if (child not already visited) then
                enqueue(child, Open);
```

https://courses.cs.washington.edu/courses/cse326/08wi/a/lectures/lecture13.pdf

Algorithm Search

Initialize as follows:
    unmark all nodes in N;
    mark node s;
    pred(s) = 0;   {that is, it has no predecessor}
    LIST = {s}
while LIST ≠ ø do
    select a node i in LIST;
    if node i is **incident** to an admissible arc (i,j) then
        mark node j;
        pred(j) := i;
        add node j to the end of LIST;
    else
        delete node i from LIST

https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

Algorithm Search

```
Initialize as follows:
    unmark all nodes in N;
    mark node s;
    pred(s) = 0;   {that is, it has no predecessor}
    LIST = {s}
while LIST ≠ ø do
    select a node i in LIST;
    if node i is incident to an admissible arc (i,j) then
        mark node j;
        pred(j) := i;
        add node j to the end of LIST;
    else
        delete node i from LIST
```

DFS : select the last node i in LIST;
BFS : select the first node i in LIST;

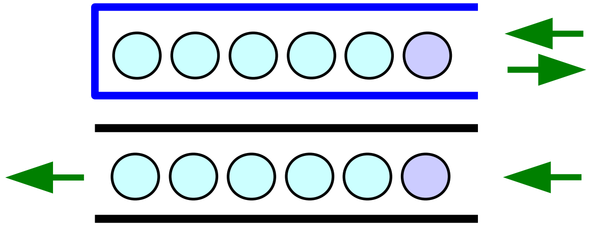
https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

Algorithm Search

Initialize as follows:
  unmark all nodes in N;
  mark node s;
  pred(s) = 0; {that is, it has no predecessor}
  LIST = {s}
while LIST ≠ ø do
  select a node i in LIST;
  if node i is **incident** to an **admissible** arc (i,j) then
    mark node j;
    pred(j) := i;
    add node j to the end of LIST;
  else
    delete node i from LIST

DFS : select the last node i in LIST;

BFS : select the first node i in LIST;

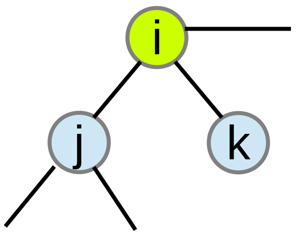


https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

Algorithm Search

pred(j) is a node that **precedes** j on some path from s;

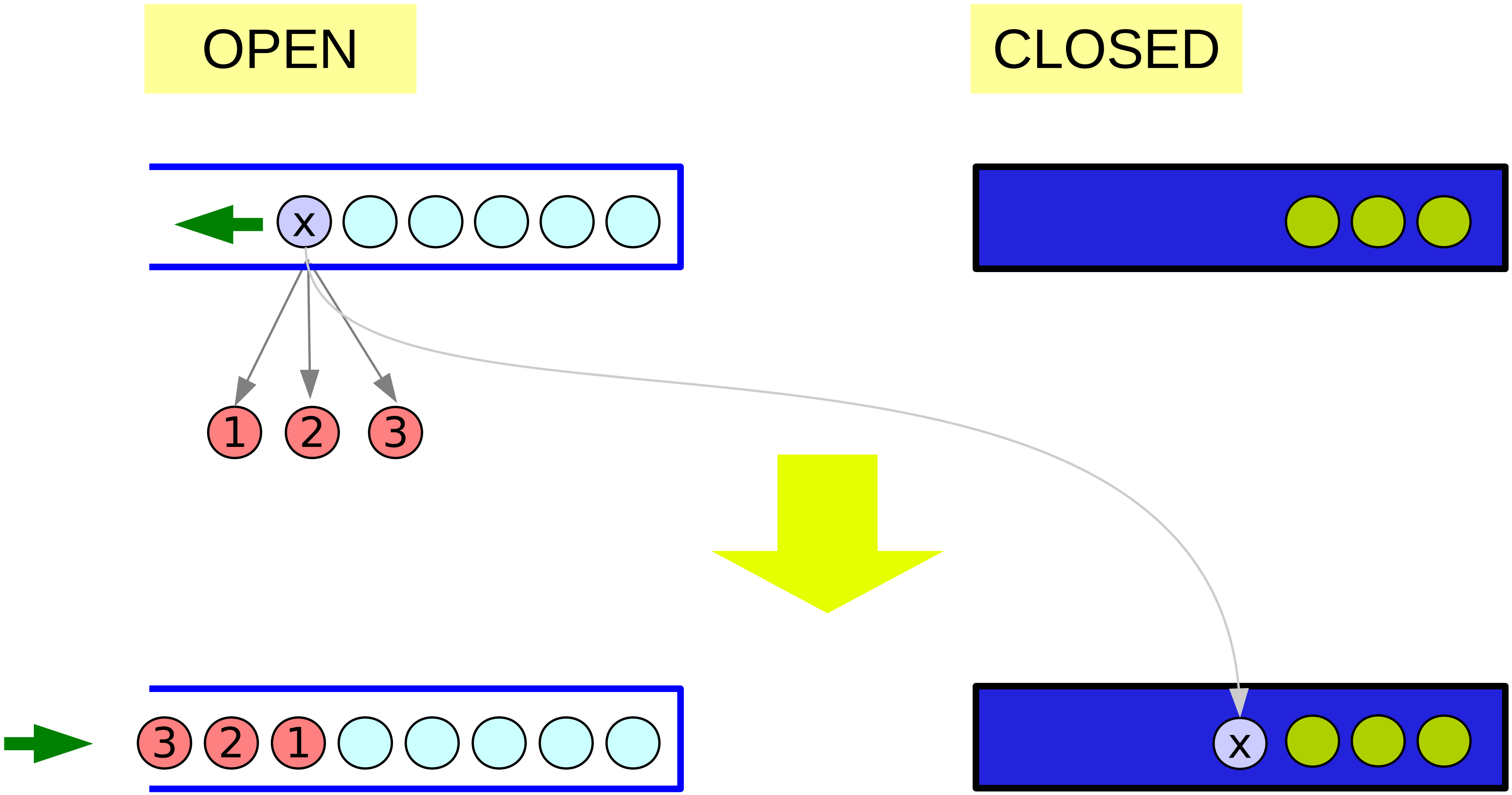
A node is either **marked** or **unmarked**.
Initially only node s is marked.
If a node is marked, it is **reachable** from node s.
An arc (i,j) ∈ A is **admissible**
if node i is marked and j is not.



https://ocw.mit.edu/courses/sloan-school-of-management/15-082j-network-optimization-fall-2010/lecture-notes/MIT15_082JF10_lec03.pdf

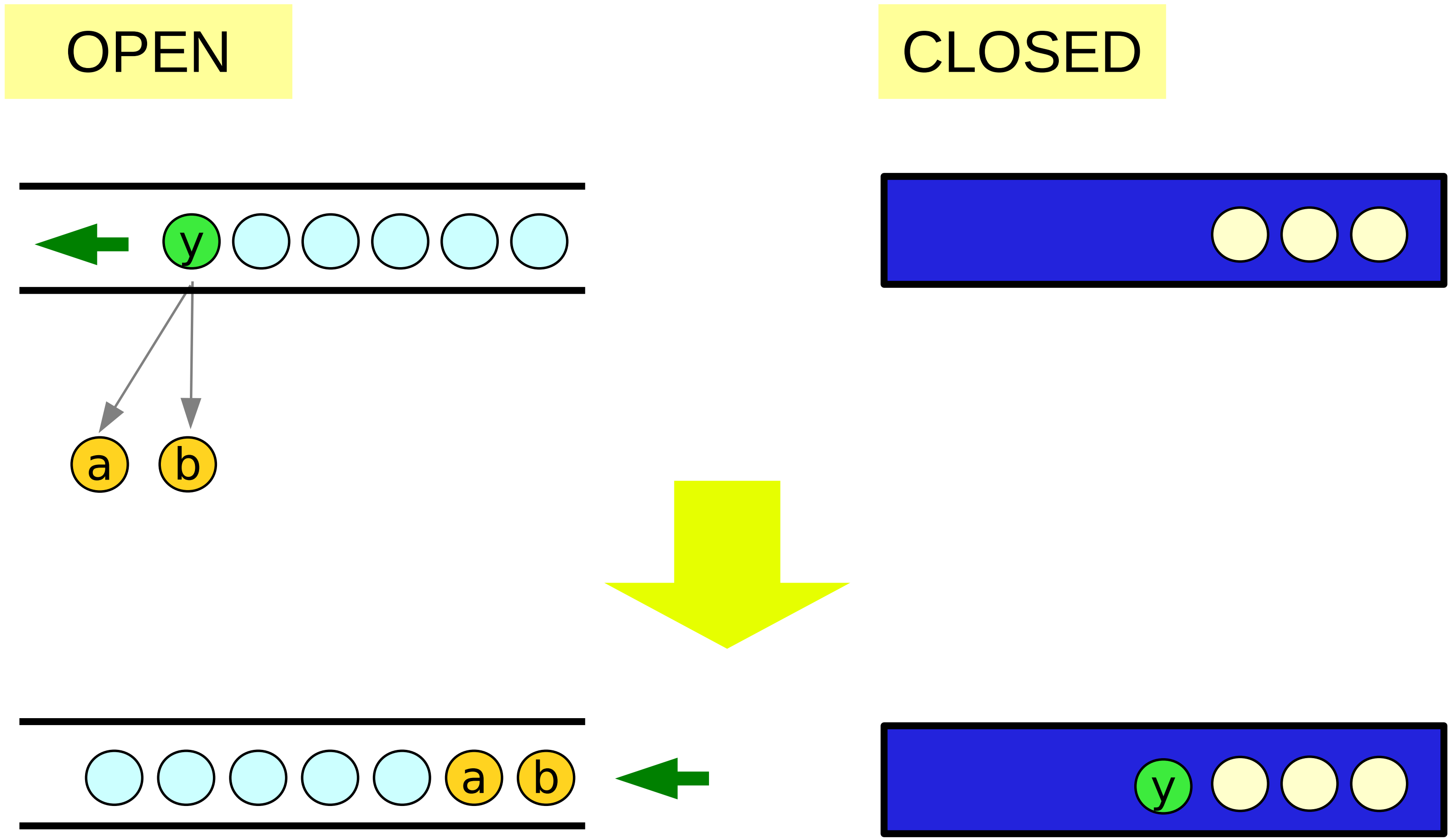
DFS

https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid



BFS

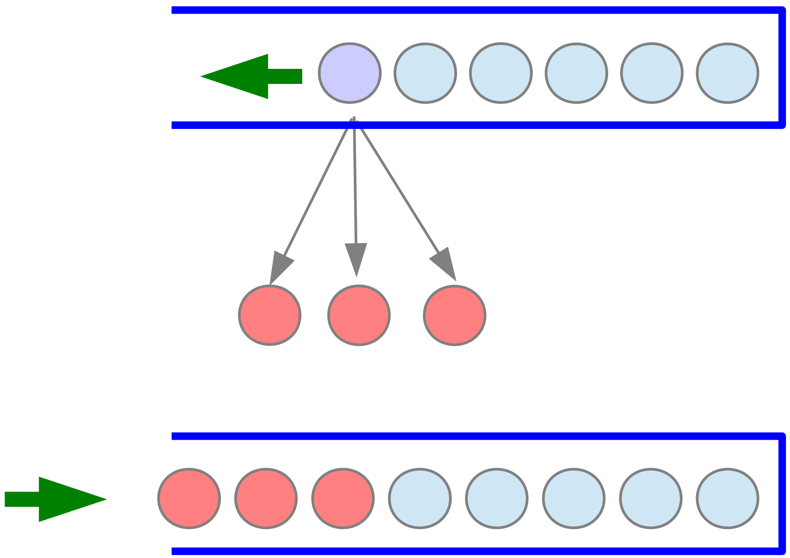
https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid



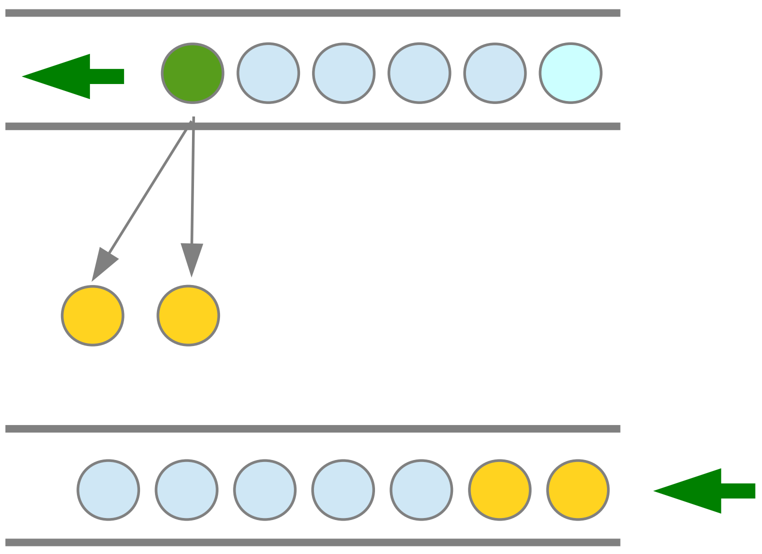
Expand Function

https://en.wikiversity.org/wiki/Artificial_intelligence/Lecture_aid

DFS (Depth First Search) | BFS (Breadth First Search)



Stack



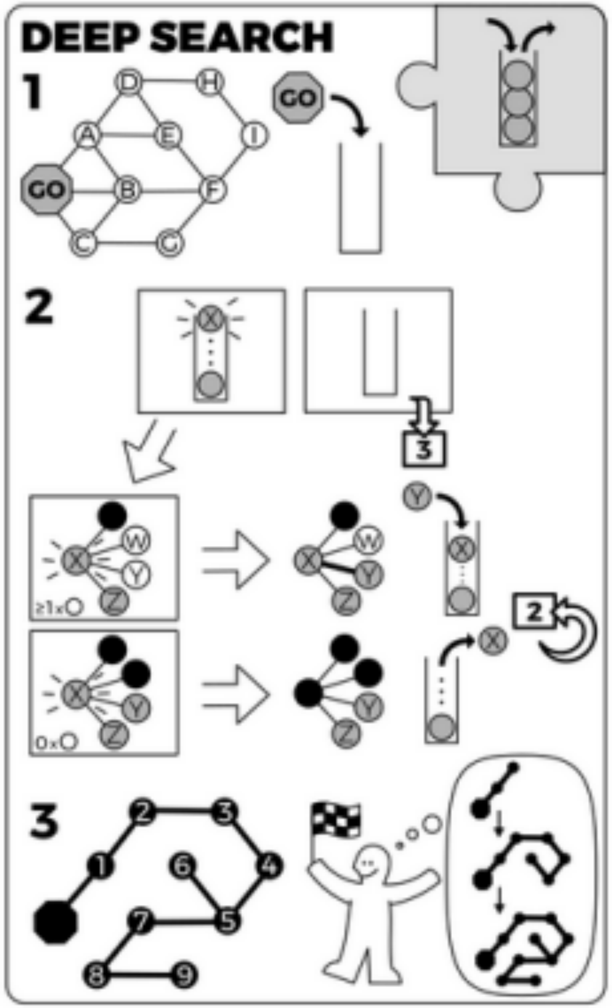
Queue

DFS Pseudocode

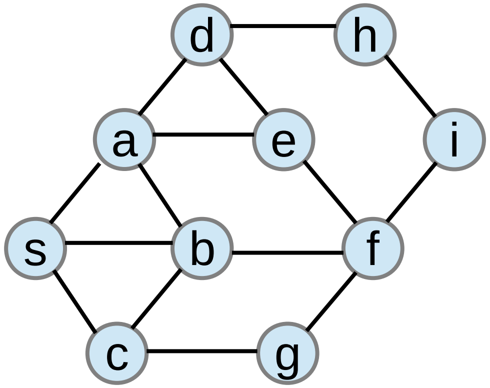
```
procedure DFS(G, v):
    label v as explored
    for all edges e in G.incidentEdges(v) do
        if edge e is unexplored then
            w ← G.adjacentVertex(v, e)
            if vertex w is unexplored then
                label e as a discovered edge
                recursively call DFS(G, w)
            else
                label e as a back edge
```

https://en.wikipedia.org/wiki/Graph_traversal

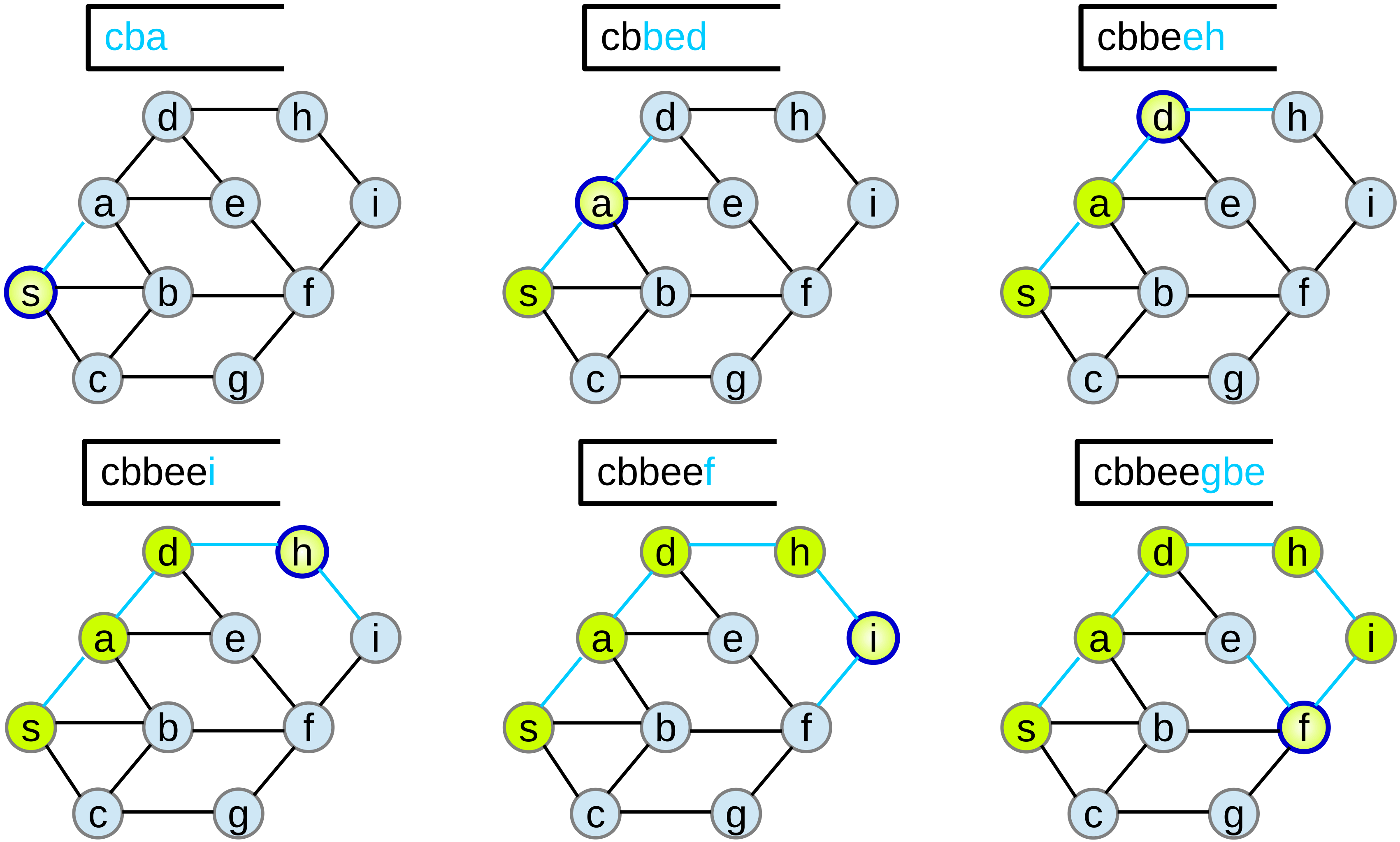
Depth First Search Example



https://en.wikipedia.org/wiki/Graph_traversal

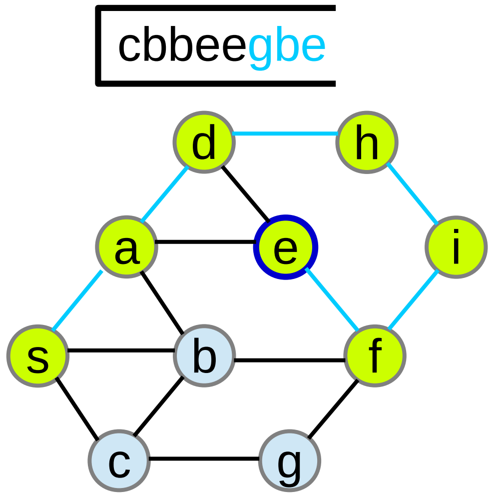


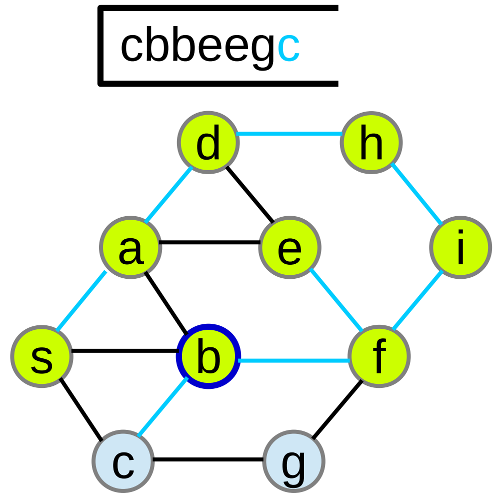
Depth First Search Example

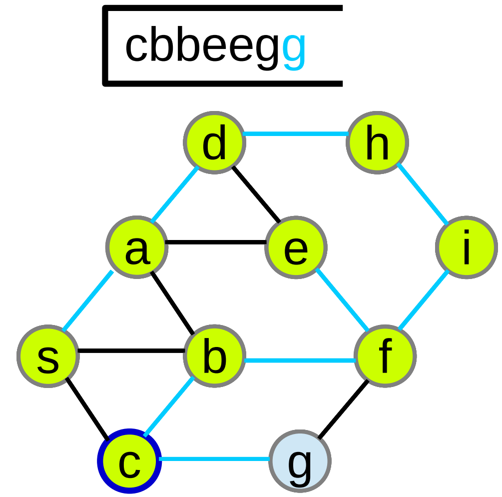


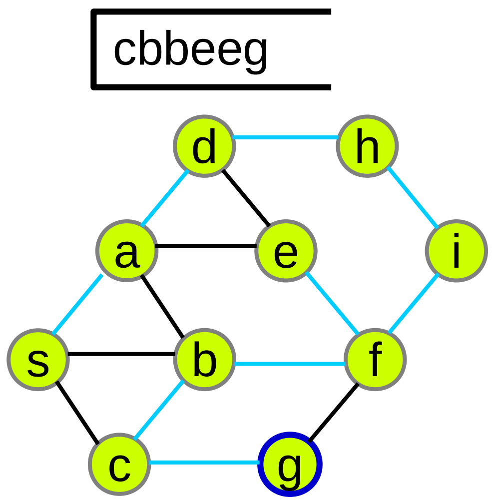
https://en.wikipedia.org/wiki/Graph_traversal

Depth First Search Example









https://en.wikipedia.org/wiki/Graph_traversal

DFS

A depth-first search (DFS)
is an algorithm for traversing a finite graph.

DFS visits the **child vertices**
before visiting the **sibling vertices**;

that is, it traverses the **depth** of any particular path
before exploring its breadth.

A **stack** (often the program's call stack via recursion) is
generally used when implementing the algorithm.

https://en.wikipedia.org/wiki/Graph_traversal

DFS Backtrack

The algorithm begins with a chosen "**root**" vertex;

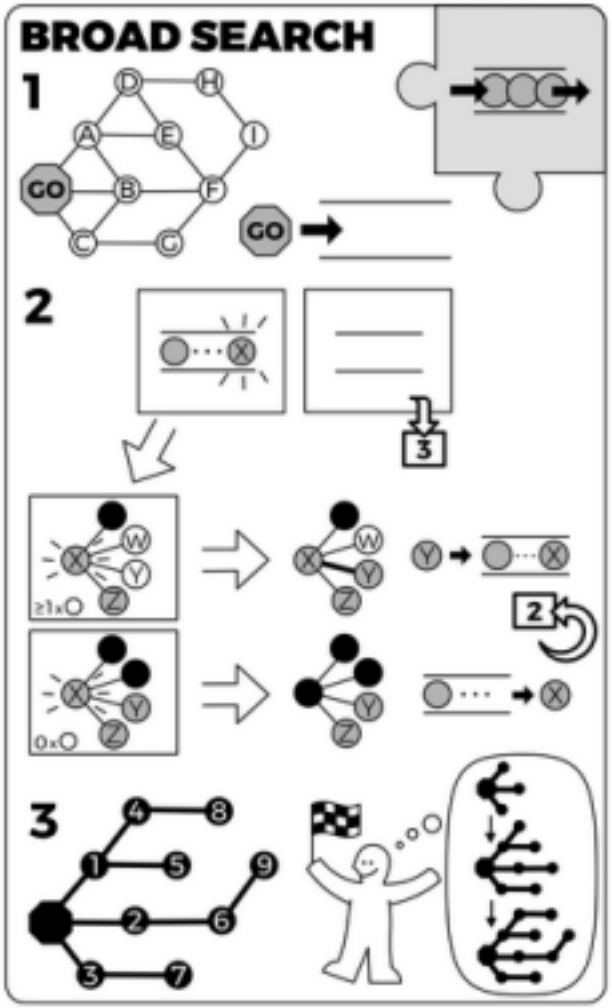
it then iteratively transitions from the **current** vertex to an **adjacent**, **unvisited** vertex, until it can no longer find an unexplored vertex to transition to from its current location.

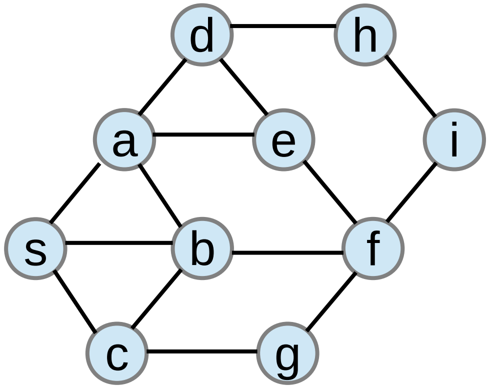
The algorithm then **backtracks** along previously visited vertices, until it finds a vertex connected to yet more uncharted territory.

It will then proceed down the new path as it had before, backtracking as it encounters **dead-ends**, and ending only when the algorithm has backtracked past the original "root" vertex from the very first step.

https://en.wikipedia.org/wiki/Graph_traversal

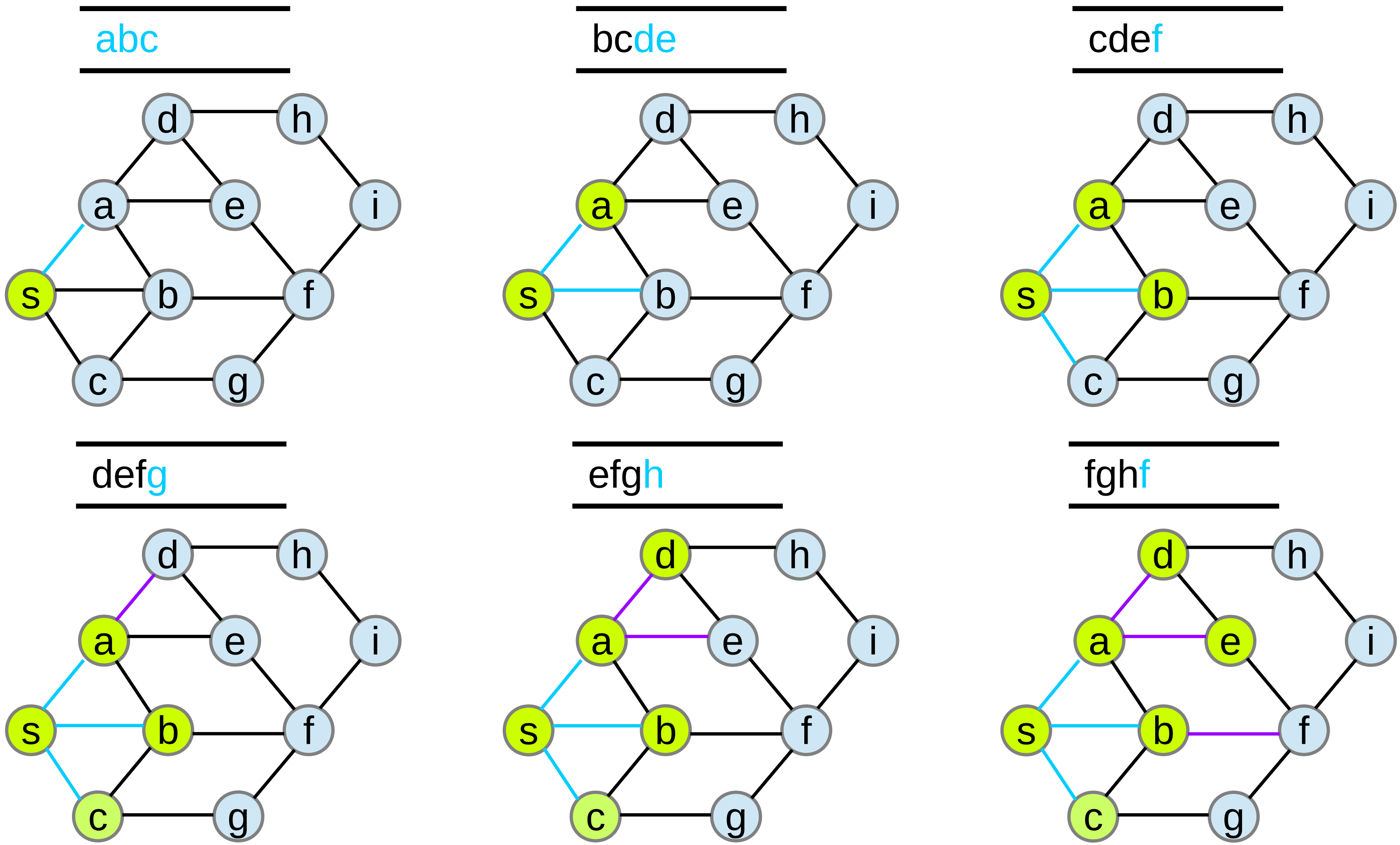
Breadth First Search Example





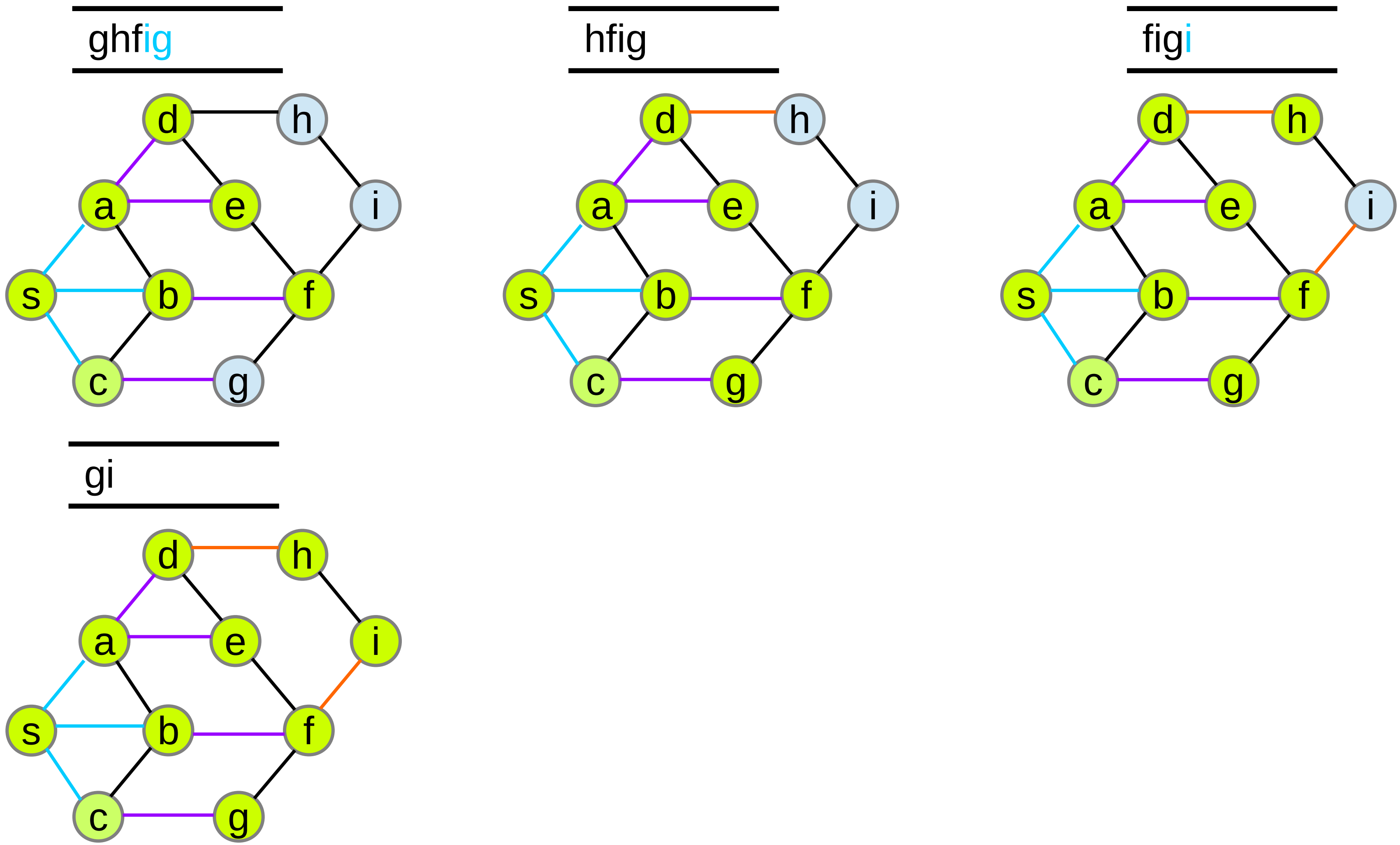
https://en.wikipedia.org/wiki/Graph_traversal

Breadth First Search Example



https://en.wikipedia.org/wiki/Graph_traversal

Breadth First Search Example



https://en.wikipedia.org/wiki/Graph_traversal

BFS

A breadth-first search (BFS) is another technique for traversing a finite graph.

BFS visits the **neighbor** vertices before visiting the **child** vertices

a **queue** is used in the search process

This algorithm is often used to find the **shortest path** from one vertex to another.

https://en.wikipedia.org/wiki/Graph_traversal

BFS Pseudocode

```
procedure BFS(G, v):
    create a queue Q
    enqueue v onto Q
    mark v
    while Q is not empty:
        t ← Q.dequeue()
        if t is what we are looking for:
            return t
        for all edges e in G.adjacentEdges(t) do
            o ← G.adjacentVertex(t, e)
            if o is not marked:
                mark o
                enqueue o onto Q
    return null
```

https://en.wikipedia.org/wiki/Graph_traversal

References

[1] http://en.wikipedia.org/
[2]

Binary Search Tree (2A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Binary Search Tree

Bnary search trees (BST),
ordered binary trees
sorted binary trees

are a particular type of container:
data structures that store "items"
(such as numbers, names etc.) in memory.

They allow fast lookup, addition and removal of items
can be used to implement either dynamic sets of items
lookup tables that allow finding an item by its key
(e.g., finding the phone number of a person by name).

https://en.wikipedia.org/wiki/Binary_search_tree

Binary Search Tree

keep their keys in sorted order
lookup operations can use the principle of binary search

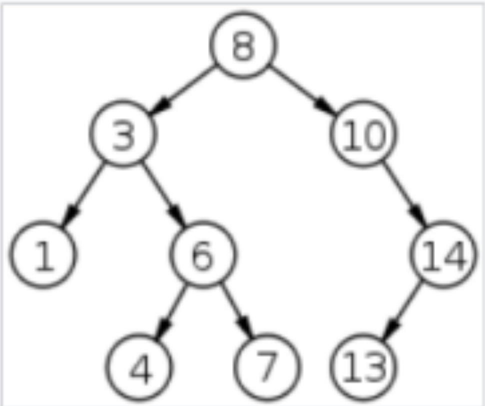
when looking for a key in a tree
or looking for a place to insert a new key,
they traverse the tree from root to leaf,
making comparisons to keys stored in the nodes
Deciding to continue in the left or right subtrees,
on the basis of the comparison.

allowing to skip searching half of the tree
each operation (lookup, insertion or deletion)
takes time proportional to log n

much better than the linear time
but slower than the corresponding operations on hash tables.

https://en.wikipedia.org/wiki/Binary_search_tree

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

$3 < 8 < 10$

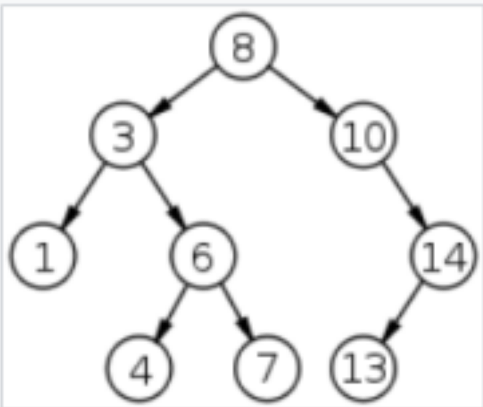
$1 < 3 < 6$ $\quad$ $10 < 14$

$4 < 6 < 7$ $\quad$ $13 < 14$

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

$1, 3, 4, 6, 7 < 8 < 10, 13, 14$

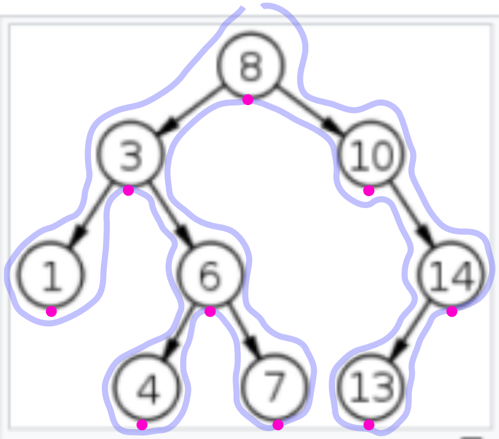
$1 < 3 < 4, 6, 7$ $\quad$ $10 < 13, 14$

$4 < 6 < 7$ $\quad$ $13 < 14$

$1, \; 3, \; 4, \; 6, \; 7, \; 8, \; 10, \; 13, \; 14$

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

$1, 3, 4, 6, 7 < 8 < 10, 13, 14$

$1 < 3 < 4, 6, 7$

$10 < 13, 14$

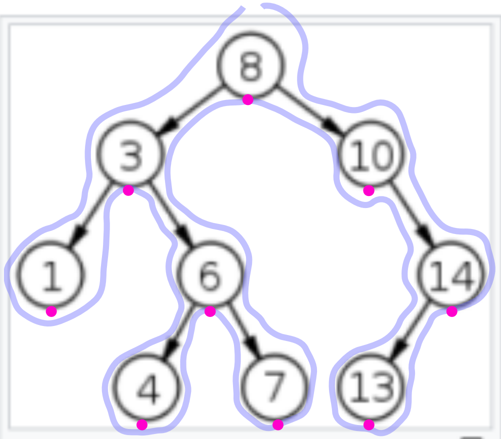
$4 < 6 < 7$

$13 < 14$

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

$1, 3, 4, 6, 7 < 8 < 10, 13, 14$

$1 < 3 < 4, 6, 7$

$10 < 13, 14$

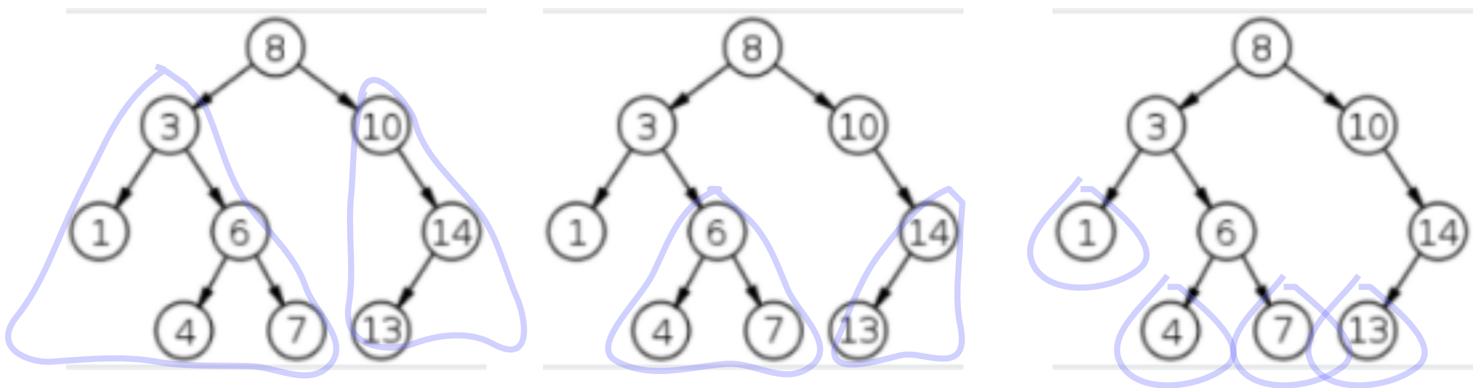
$4 < 6 < 7$

$13 < 14$

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

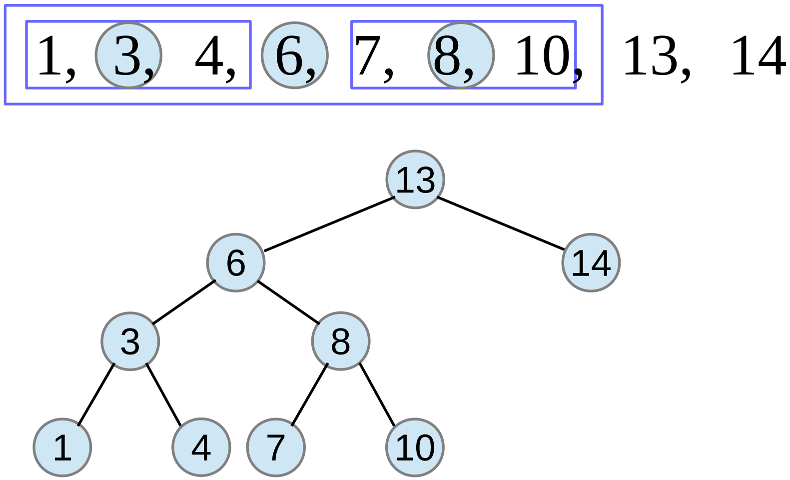
Infix, Prefix, Postfix Notations

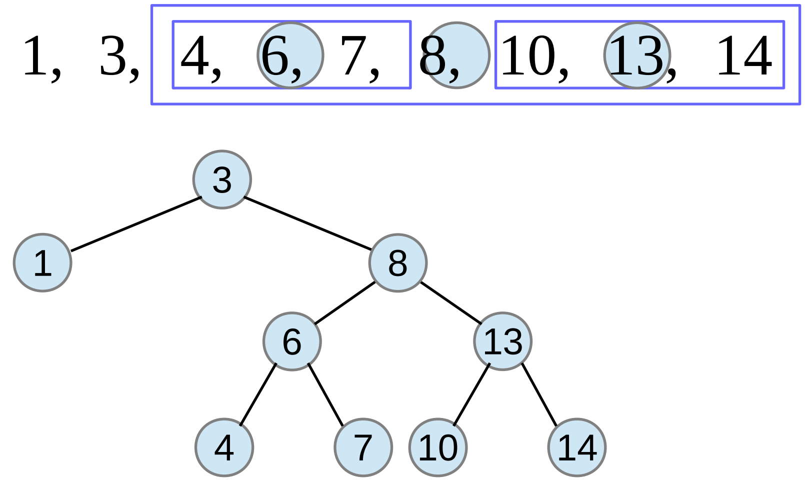


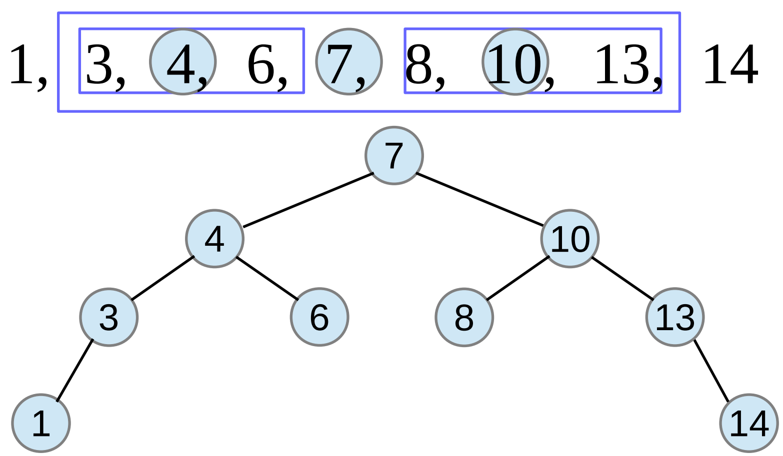
1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Infix, Prefix, Postfix Notations

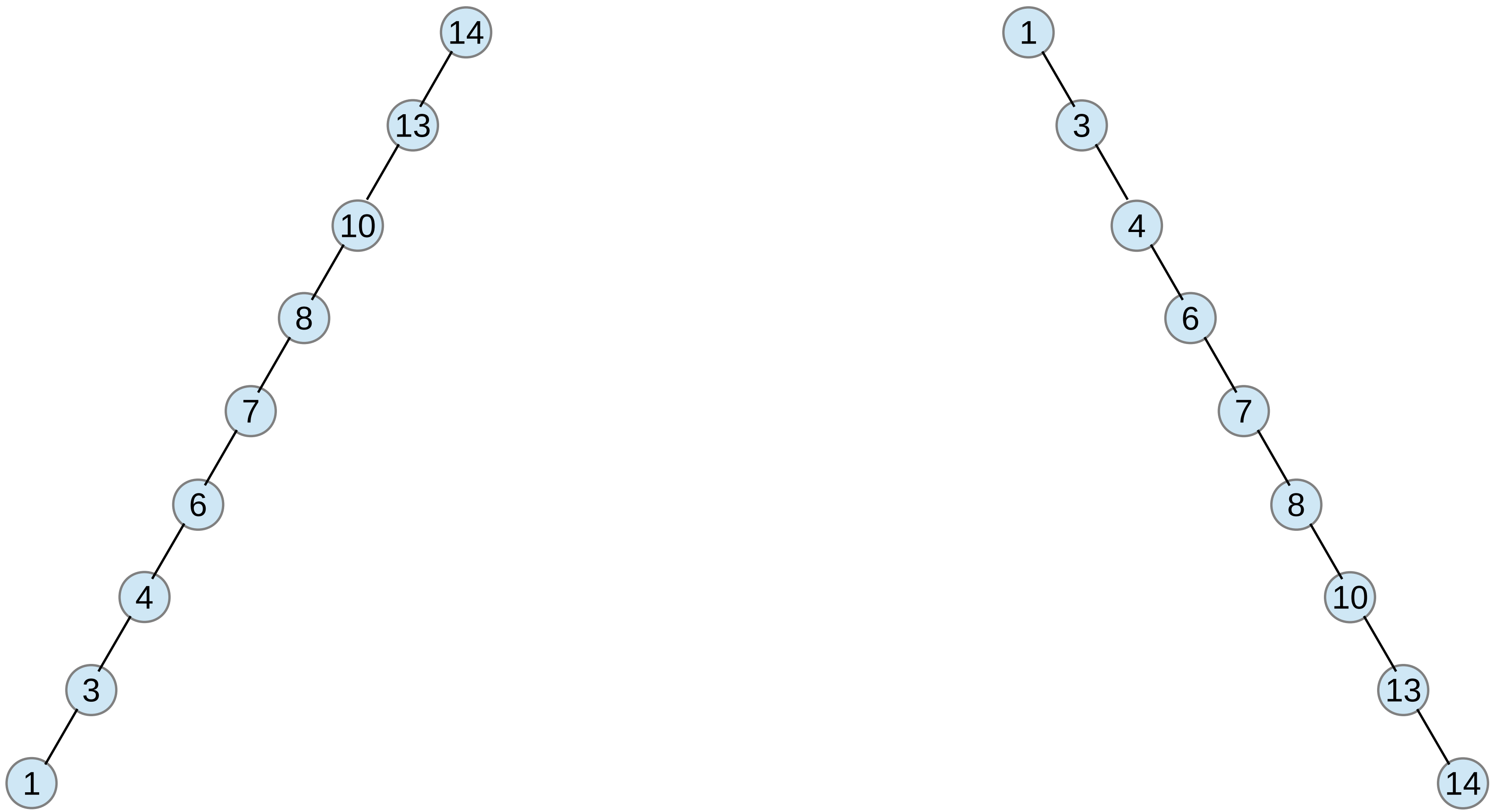




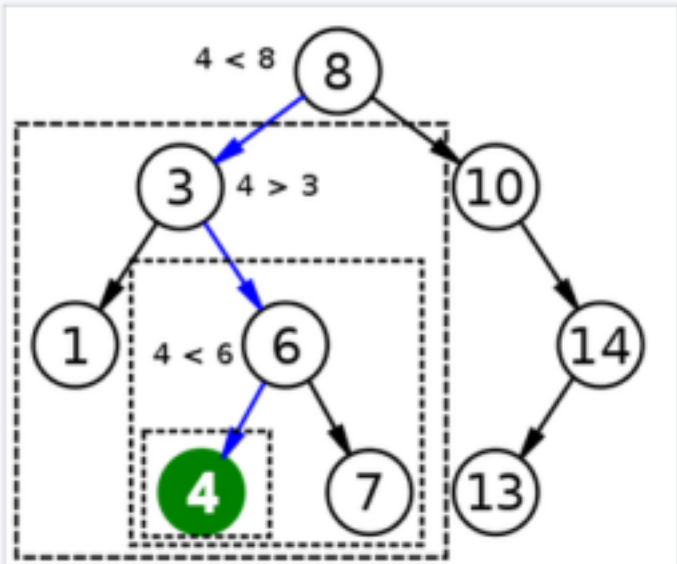


Infix, Prefix, Postfix Notations

1, 3, 4, 6, 7, 8, 10, 13, 14 1, 3, 4, 6, 7, 8, 10, 13, 14



Binary Search



Binary search trees are searched using an algorithm similar to binary search.

https://en.wikipedia.org/wiki/Binary_search_algorithm

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

https://en.wikipedia.org/wiki/Morphism

Deletion

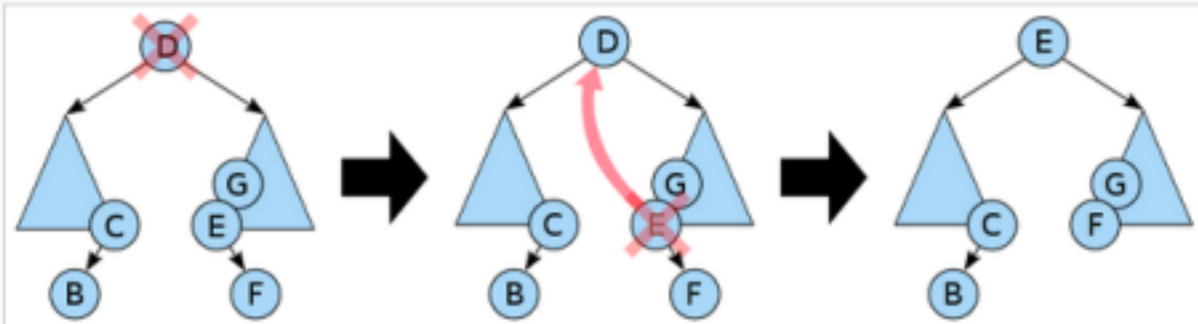
1. Deleting a node with no children:
   simply remove the node from the tree.

2. Deleting a node with one child:
   remove the node and replace it with its child.

3. Deleting a node with two children:
   call the node to be deleted D.
   Do not delete D.
   Instead, choose either its in-order predecessor node
   or its in-order successor node as replacement node E.
   Copy the user values of E to D
   If E does not have a child
   simply remove E from its previous parent G.
   If E has a child, say F, it is a right child.
   Replace E with F at E's parent.

https://en.wikipedia.org/wiki/Morphism

Deletion



Deleting a node with two children from a binary search tree. First the leftmost node in the right subtree, the in-order successor *E*, is identified. Its value is copied into the node *D* being deleted. The in-order successor can then be easily deleted because it has at most one child. The same method works symmetrically using the in-order predecessor *C*.

https://en.wikipedia.org/wiki/Morphism

References

[1] http://en.wikipedia.org/
[2]

Binary Search Tree (2A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Binary Search Tree

Bnary search trees (BST),
ordered binary trees
sorted binary trees

are a particular type of container:
data structures that store "items"
(such as numbers, names etc.) in memory.

They allow fast lookup, addition and removal of items
can be used to implement either dynamic sets of items
lookup tables that allow finding an item by its key
(e.g., finding the phone number of a person by name).

https://en.wikipedia.org/wiki/Binary_search_tree

Binary Search Tree

keep their keys in sorted order
lookup operations can use the principle of binary search

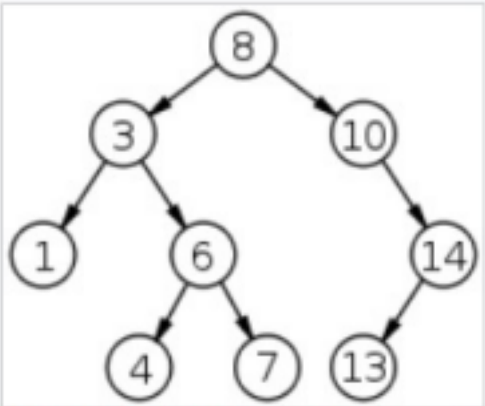
when looking for a key in a tree
or looking for a place to insert a new key,
they traverse the tree from root to leaf,
making comparisons to keys stored in the nodes
Deciding to continue in the left or right subtrees,
on the basis of the comparison.

allowing to skip searching half of the tree
each operation (lookup, insertion or deletion)
takes time proportional to log n

much better than the linear time
but slower than the corresponding operations on hash tables.

https://en.wikipedia.org/wiki/Binary_search_tree

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

3 < 8 < 10

1 < 3 < 6

10 < 14

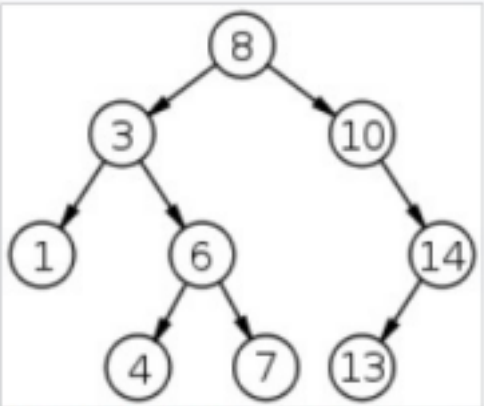
4 < 6 < 7

13 < 14

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

$1, 3, 4, 6, 7 < 8 < 10, 13, 14$

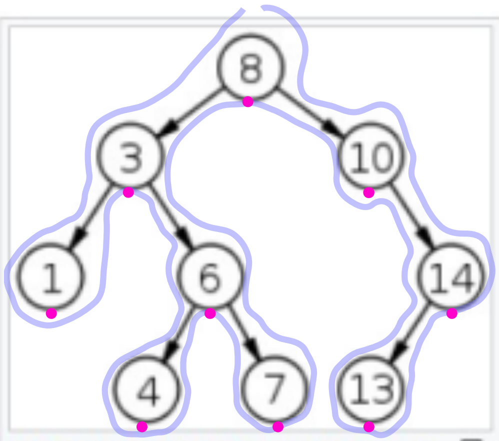
$1 < 3 < 4, 6, 7$ $\quad$ $10 < 13, 14$

$4 < 6 < 7$ $\quad$ $13 < 14$

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Infix, Prefix, Postfix Notations



A binary search tree of size 9 and depth 3, with 8 at the root. The leaves are not drawn.

$1, 3, 4, 6, 7 < 8 < 10, 13, 14$

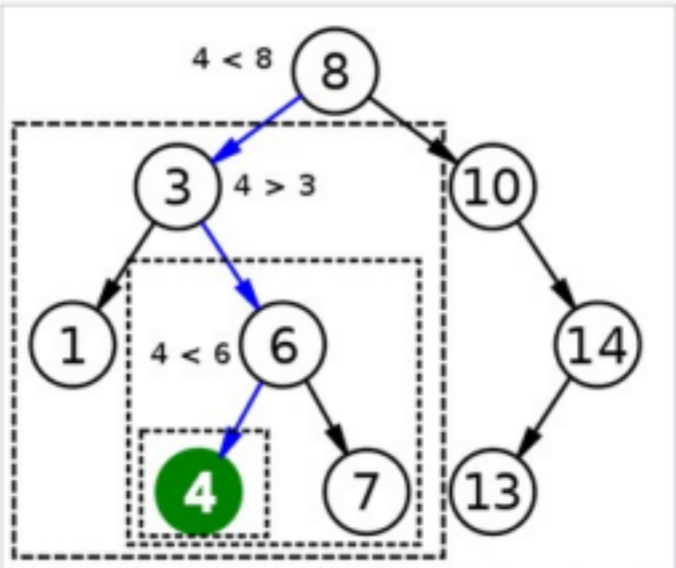
$1 < 3 < 4, 6, 7$ $10 < 13, 14$

$4 < 6 < 7$ $13 < 14$

1, 3, 4, 6, 7, 8, 10, 13, 14

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

Binary Search



Binary search trees are searched using an algorithm similar to binary search.

https://en.wikipedia.org/wiki/Binary_search_algorithm

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

https://en.wikipedia.org/wiki/Morphism

Deletion

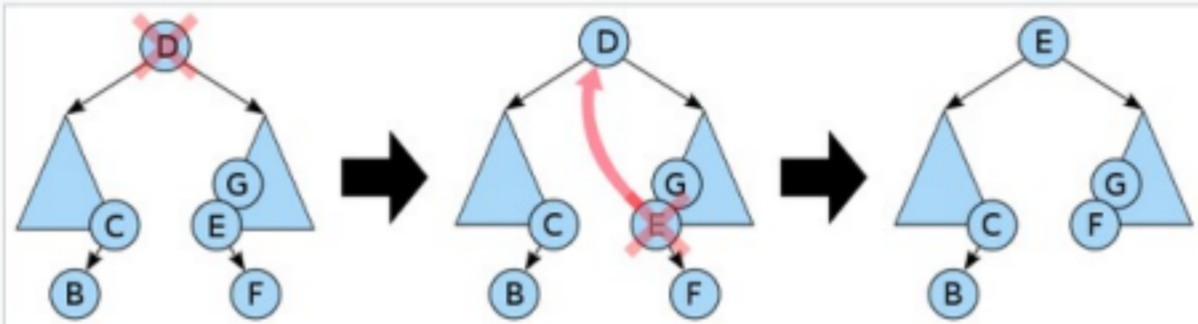
1. Deleting a node with no children:
   simply remove the node from the tree.

2. Deleting a node with one child:
   remove the node and replace it with its child.

3. Deleting a node with two children:
   call the node to be deleted D.
   Do not delete D.
   Instead, choose either its in-order predecessor node
   or its in-order successor node as replacement node E.
   Copy the user values of E to D
   If E does not have a child
      simply remove E from its previous parent G.
   If E has a child, say F, it is a right child.
      Replace E with F at E's parent.

https://en.wikipedia.org/wiki/Morphism

Deletion



Deleting a node with two children from a binary search tree. First the leftmost node in the right subtree, the in-order successor *E*, is identified. Its value is copied into the node *D* being deleted. The in-order successor can then be easily deleted because it has at most one child. The same method works symmetrically using the in-order predecessor *C*.

https://en.wikipedia.org/wiki/Morphism

References

[1] http://en.wikipedia.org/
[2]